API reference for labbench

Release 0.20

Dan Kuester (NIST)

Jun 26, 2019

Contents

CHAPTER 1

labbench.backends module

class labbench.backends.**CommandLineWrapper**(*resource=None*, ***settings*)

Bases: *labbench.core.Device*

Virtual device representing for interacting with a command line executable. It supports threaded data logging through standard input, standard output, and standard error pipes.

On connection, the *backend* attribute is None. On a call to execute(), *backend* becomes is a subprocess instance. When EOF is reached on the executable's stdout, the backend is assumed terminated and is reset to None.

When *execute* is called, the program runs in a subprocess. The output piped to the command line standard output is queued in a background thread. Call read_stdout() to retreive (and clear) this queued stdout.

Parameters

- **arguments** (List *()*) – list of command line arguments to pass into the executable
- **binary_path** (Unicode *()*) – path to the file to run
- **resource** (Unicode *()*) – Addressing information needed to make a connection to a device. Type and format are determined by the subclass implementation
- **timeout** (Float *(min=0, max=inf)*) – Timeout (sec) after disconnect is called before killing the process

background(**extra_arguments*, ***flags*)

Run the executable in the background (returning immediately while the executable continues running).

Once the background process is running,

- Retreive standard output from the executable with self.read_stdout
- Write to standard input self.write_stdin
- Kill the process with self.kill
- Check whether the process is running with self.running

Normally, the command line arguments are determined by

- appending extra_arguments to the global arguments in self.settings.arguments, and

- appending pairs of [key,value] from the *flags* dictionary to the global flags defined with command flags in local state traits in *self.settings*

Use the self.no_state_arguments context manager to skip these global arguments like this:

```
with self.no_state_arguments:
    self.background(...)
```

Returns None

clear()
Clear queued standard output, discarding any contents

connect()
The *connect()* method exists to comply with the Device object protocol. Call the execute() method when connected to execute the binary.

disconnect()
Backend implementations must overload this to disconnect an existing connection to the resource encapsulated in the object.

exception_on_stderr
Use this context manager to raise exceptions if a process outputs to standard error during background execution.

foreground(**extra_arguments*, ***flags*)
Blocking execution of the binary at the file location *self.settings.binary_path*.

Normally, the command line arguments are determined by * appending extra_arguments to the global arguments in self.settings.arguments, and * appending pairs of [key,value] from the *flags* dictionary to the

global flags defined with command flags in local state traits in *self.settings*

Use the self.no_state_arguments context manager to skip these global arguments like this:

```
with self.no_state_arguments:
    self.foreground(...)
```

Returns the return code of the process after its completion

kill()
If a process is running in the background, kill it. Sends a logger warning if no process is running.

no_state_arguments
Use this context manager to disable automatic use of state traits in generating argument strings.

read_stdout(*wait_for=0*)
Return string output queued from stdout for a process running in the background. This clears the queue.

Returns an empty string if the command line program has not been executed or is empty. Running the command line multiple times overwrites the queue.

Returns stdout

respawn
Use this context manager to respawning background execution.

running()
Return whether the executable is currently running

Returns True if running, otherwise False

class settings(*device*, **args*, ***kws*)

Bases: `labbench.core.settings`

Container for settings traits in a Device.

These settings are stored only on the host; setting or getting these values do not trigger live updates (or any communication) with the device. These define connection addressing information, communication settings, and options that only apply to implementing python support for the device.

The device uses this container to define the keyword options supported by its __init__ function. These are applied when you instantiate the device. After you instantiate the device, you can still change the setting with:

```
Device.settings.resource = 'insert-your-address-string-here'
```

trait attributes:

- *arguments*: *List*
- *arguments_min*: *Int*
- *binary_path*: *Unicode*
- *concurrency_support*: *Bool*
- *resource*: *Unicode*
- *timeout*: *Float*

arguments

List()

list of command line arguments to pass into the executable

arguments_min

Int(min=0,read_only=True)

minimum number of extra command line arguments to pass to the executable

binary_path

Unicode()

path to the file to run

concurrency_support

Bool(read_only=True)

Whether this backend supports threading

classmethod define(***kws*)

Change default values of the settings in parent settings, without redefining the full class. redefined according to each keyword argument. For example:

```
MyInstrumentClass.settings.define(parameter=7)
```

changes the default value of the *parameter* setting in *MyInstrumentClass.settings* to *7*. This is a convenience function to avoid completely redefining *parameter* if it was defined in a parent class of *MyInstrumentClass*.

resource

Unicode()

Addressing information needed to make a connection to a device. Type and format are determined by the subclass implementation

timeout
Float(min=0,max=inf)

Timeout (sec) after disconnect is called before killing the process

class state(*device*, **args*, ***kws*)
Bases: `labbench.core.HasStateTraits`

Container for state traits in a Device. Getting or setting state traits triggers live updates: communication with the device to get or set the value on the Device. Therefore, getting or setting state traits needs the device to be connected.

To set a state value inside the device, use normal python assigment:

```
device.state.parameter = value
```

To get a state value from the device, you can also use it as a normal python variable:

```
variable = device.state.parameter + 1
```

trait attributes:

- *connected*: *Bool*

connected
Bool(read_only=True)

whether the `Device` instance is connected

classmethod getter(*func*)
Use this as a decorator to define a setter function for all traits in this class. The getter should take one argument: the instance of the trait to get. It should perform any operation needed to retrieve the current value of the device state corresponding to the supplied trait, using *self._device*.

One example is to send a command defined by trait.command.

The function should return a value that is the state from the device.

A trait that has its own getter defined will ignore this one.

classmethod setter(*func*)
Use this as a decorator to define a setter function for all traits in this class. The setter should take two arguments: the instance of the trait to get, and the value to set. It should perform any operation needed to apply the given value to the trait's state in *self._device*. One example is to send a command defined by trait.command.

Any return value from the function is ignored.

A trait that has its own setter defined will ignore this one.

write_stdin(*text*)
Write characters to stdin if a background process is running. Raises Exception if no background process is running.

class labbench.backends.**DotNetDevice**(*resource=None*, ***settings*)
Bases: *labbench.core.Device*

This Device backend represents a wrapper around a .NET library. It is implemented with pythonnet, and handlesimports.

In order to implement a DotNetDevice subclass:

- define the attribute *library* = <*mypythonmodule.wheredllbinariesare*>, the python module with copies of the .NET DLLs are

- define the attribute *dll_name* = *"mydllname.dll"*, the name of the DLL binary in the python module above

When a DotNetDevice is instantiated, it tries to load the dll according to the above specifications.

Other attributes of DotNetDevice use the following conventions

- *backend* may be set by a subclass *connect* method (otherwise it is left as None)

Parameters **resource** (Unicode()) – Addressing information needed to make a connection to a device. Type and format are determined by the subclass implementation

connect()
Backend implementations overload this to open a backend connection to the resource.

disconnect()
Backend implementations must overload this to disconnect an existing connection to the resource encapsulated in the object.

class settings(*device*, **args*, ***kws*)
Bases: labbench.core.HasSettingsTraits

Container for settings traits in a Device.

These settings are stored only on the host; setting or getting these values do not trigger live updates (or any communication) with the device. These define connection addressing information, communication settings, and options that only apply to implementing python support for the device.

The device uses this container to define the keyword options supported by its __init__ function. These are applied when you instantiate the device. After you instantiate the device, you can still change the setting with:

```
Device.settings.resource = 'insert-your-address-string-here'
```

trait attributes:

- *concurrency_support*: *Bool*
- *resource*: *Unicode*

concurrency_support
Bool(read_only=True)

Whether this backend supports threading

classmethod define(***kws*)
Change default values of the settings in parent settings, without redefining the full class. redefined according to each keyword argument. For example:

```
MyInstrumentClass.settings.define(parameter=7)
```

changes the default value of the *parameter* setting in *MyInstrumentClass.settings* to *7*. This is a convenience function to avoid completely redefining *parameter* if it was defined in a parent class of *MyInstrumentClass*.

resource
Unicode()

Addressing information needed to make a connection to a device. Type and format are determined by the subclass implementation

class state(*device*, **args*, ***kws*)
Bases: labbench.core.HasStateTraits

Container for state traits in a Device. Getting or setting state traits triggers live updates: communication with the device to get or set the value on the Device. Therefore, getting or setting state traits needs the device to be connected.

To set a state value inside the device, use normal python assigment:

```
device.state.parameter = value
```

To get a state value from the device, you can also use it as a normal python variable:

```
variable = device.state.parameter + 1
```

trait attributes:

- *connected*: *Bool*

connected
Bool(read_only=True)

whether the `Device` instance is connected

classmethod getter(*func*)
Use this as a decorator to define a setter function for all traits in this class. The getter should take one argument: the instance of the trait to get. It should perform any operation needed to retrieve the current value of the device state corresponding to the supplied trait, using *self._device*.

One example is to send a command defined by trait.command.

The function should return a value that is the state from the device.

A trait that has its own getter defined will ignore this one.

classmethod setter(*func*)
Use this as a decorator to define a setter function for all traits in this class. The setter should take two arguments: the instance of the trait to get, and the value to set. It should perform any operation needed to apply the given value to the trait's state in *self._device*. One example is to send a command defined by trait.command.

Any return value from the function is ignored.

A trait that has its own setter defined will ignore this one.

class labbench.backends.**EmulatedVISADevice**(*resource=None*, ***settings*)
Bases: *labbench.core.Device*

Act as a VISA device without dispatching any visa commands

Parameters **resource** (*Unicode()*) – Addressing information needed to make a connection to a device. Type and format are determined by the subclass implementation

connect()
Backend implementations overload this to open a backend connection to the resource.

disconnect()
Backend implementations must overload this to disconnect an existing connection to the resource encapsulated in the object.

classmethod set_backend(*backend_name*)
backend_name can be 'py' or 'ni'

class settings(*device*, **args*, ***kws*)
Bases: labbench.backends.settings

Container for settings traits in a Device.

These settings are stored only on the host; setting or getting these values do not trigger live updates (or any communication) with the device. These define connection addressing information, communication settings, and options that only apply to implementing python support for the device.

The device uses this container to define the keyword options supported by its __init__ function. These are applied when you instantiate the device. After you instantiate the device, you can still change the setting with:

```
Device.settings.resource = 'insert-your-address-string-here'
```

trait attributes:

- *concurrency_support*: *Bool*
- *read_termination*: *Unicode*
- *resource*: *Unicode*
- *write_termination*: *Unicode*

concurrency_support
Bool(read_only=True)

Whether this backend supports threading

classmethod define(***kws*)
Change default values of the settings in parent settings, without redefining the full class. redefined according to each keyword argument. For example:

```
MyInstrumentClass.settings.define(parameter=7)
```

changes the default value of the *parameter* setting in *MyInstrumentClass.settings* to *7*. This is a convenience function to avoid completely redefining *parameter* if it was defined in a parent class of *MyInstrumentClass*.

read_termination
Unicode(read_only='connected')

termination character to indicate end of message on receive from the instrument

resource
Unicode()

Addressing information needed to make a connection to a device. Type and format are determined by the subclass implementation

write_termination
Unicode(read_only='connected')

termination character to indicate end of message in messages sent to the instrument

class state(*device*, **args*, ***kws*)
Bases: `labbench.backends.state`

Container for state traits in a Device. Getting or setting state traits triggers live updates: communication with the device to get or set the value on the Device. Therefore, getting or setting state traits needs the device to be connected.

To set a state value inside the device, use normal python assigment:

```
device.state.parameter = value
```

To get a state value from the device, you can also use it as a normal python variable:

```
variable = device.state.parameter + 1
```

trait attributes:

- *connected*: *Bool*
- *identity*: *Unicode*
- *options*: *Unicode*
- *status_byte*: *Dict*

connected
Bool(read_only=True)

whether the `Device` instance is connected

classmethod getter(*func*)
Use this as a decorator to define a setter function for all traits in this class. The getter should take one argument: the instance of the trait to get. It should perform any operation needed to retrieve the current value of the device state corresponding to the supplied trait, using *self._device*.

One example is to send a command defined by trait.command.

The function should return a value that is the state from the device.

A trait that has its own getter defined will ignore this one.

identity
Unicode(command='*IDN',read_only=True,cache=True)

identity string reported by the instrument

options
Unicode(command='*OPT',read_only=True,cache=True)

options reported by the instrument

classmethod setter(*func*)
Use this as a decorator to define a setter function for all traits in this class. The setter should take two arguments: the instance of the trait to get, and the value to set. It should perform any operation needed to apply the given value to the trait's state in *self._device*. One example is to send a command defined by trait.command.

Any return value from the function is ignored.

A trait that has its own setter defined will ignore this one.

status_byte
Dict(command='*STB',read_only=True)

VISA status byte reported by the instrument

class labbench.backends.**LabviewSocketInterface**(*resource=None*, ***settings*)
Bases: *labbench.core.Device*

Implement the basic sockets-based control interface for labview. This implementation uses a transmit and receive socket.

State sets are implemented by simple ' command value' strings and implemented with the 'command' keyword (like VISA strings). Subclasses can therefore implement support for commands in specific labview VI the same was as in VISA commands by assigning the commands implemented in the corresponding labview VI.

The *resource* argument (which can also be set as *settings.resource*) is the ip address of the host where the labview script is running. Use the tx_port and rx_port attributes to set the TCP/IP ports where communication is to take place.

Parameters

- **delay** (Float *(min=-inf,max=inf)*) – time to wait after each state write or query
- **resource** (Unicode *()*) – IP address where the LabView VI listens for a socket
- **rx_buffer_size** (Int *()*) –
- **rx_port** (Int *()*) – TX port to send to the LabView VI
- **timeout** (Float *(min=-inf,max=inf)*) – maximum time to wait for a reply after sending before raising an Exception
- **tx_port** (Int *()*) – TX port to send to the LabView VI

clear()
Clear any data present in the read socket buffer.

connect()
Backend implementations overload this to open a backend connection to the resource.

disconnect()
Backend implementations must overload this to disconnect an existing connection to the resource encapsulated in the object.

read(*convert_func=None*)
Receive from the rx socket until *self.settings.rx_buffer_size* samples are received or timeout happens after *self.timeout* seconds.

Optionally, apply the conversion function to the value after it is received.

class settings(*device*, **args*, ***kws*)
Bases: labbench.core.settings

Container for settings traits in a Device.

These settings are stored only on the host; setting or getting these values do not trigger live updates (or any communication) with the device. These define connection addressing information, communication settings, and options that only apply to implementing python support for the device.

The device uses this container to define the keyword options supported by its __init__ function. These are applied when you instantiate the device. After you instantiate the device, you can still change the setting with:

```
Device.settings.resource = 'insert-your-address-string-here'
```

trait attributes:

- *concurrency_support*: *Bool*
- *delay*: *Float*
- *resource*: *Unicode*
- *rx_buffer_size*: *Int*
- *rx_port*: *Int*
- *timeout*: *Float*
- *tx_port*: *Int*

concurrency_support
Bool(read_only=True)

Whether this backend supports threading

classmethod define(***kws*)
Change default values of the settings in parent settings, without redefining the full class. redefined according to each keyword argument. For example:

```
MyInstrumentClass.settings.define(parameter=7)
```

changes the default value of the *parameter* setting in *MyInstrumentClass.settings* to *7*. This is a convenience function to avoid completely redefining *parameter* if it was defined in a parent class of *MyInstrumentClass*.

delay
Float(min=-inf,max=inf)

time to wait after each state write or query

resource
Unicode()

IP address where the LabView VI listens for a socket

rx_buffer_size
Int()

rx_port
Int()

TX port to send to the LabView VI

timeout
Float(min=-inf,max=inf)

maximum time to wait for a reply after sending before raising an Exception

tx_port
Int()

TX port to send to the LabView VI

class state(*device*, **args*, ***kws*)
Bases: `labbench.core.state`

Container for state traits in a Device. Getting or setting state traits triggers live updates: communication with the device to get or set the value on the Device. Therefore, getting or setting state traits needs the device to be connected.

To set a state value inside the device, use normal python assigment:

```
device.state.parameter = value
```

To get a state value from the device, you can also use it as a normal python variable:

```
variable = device.state.parameter + 1
```

trait attributes:

- *connected*: *Bool*

connected
Bool(read_only=True)

whether the `Device` instance is connected

classmethod getter(*func*)
Use this as a decorator to define a setter function for all traits in this class. The getter should take one argument: the instance of the trait to get. It should perform any operation needed to retrieve the current value of the device state corresponding to the supplied trait, using *self._device*.

One example is to send a command defined by trait.command.

The function should return a value that is the state from the device.

A trait that has its own getter defined will ignore this one.

classmethod setter(*func*)
Use this as a decorator to define a setter function for all traits in this class. The setter should take two arguments: the instance of the trait to get, and the value to set. It should perform any operation needed to apply the given value to the trait's state in *self._device*. One example is to send a command defined by trait.command.

Any return value from the function is ignored.

A trait that has its own setter defined will ignore this one.

write(*msg*)
Send a string over the tx socket.

class labbench.backends.**SerialDevice**(*resource=None*, ***settings*)
Bases: *labbench.core.Device*

A general base class for communication with serial devices. Unlike (for example) VISA instruments, there is no standardized command format like SCPI. The implementation is therefore limited to connect and disconnect, which open or close a pyserial connection object: the *link* attribute. Subclasses can read or write with the link attribute like they would any other serial instance.

A SerialDevice resource string is the same as the platform-dependent *port* argument to new serial.Serial objects.

Subclassed devices that need state descriptors will need to implement state_get and state_set methods in order to define how the state descriptors set and get operations.

Parameters

- **baud_rate** (`Int(min=1)`) – Data rate of the physical serial connection.
- **dsrdtr** (`Bool()`) – Whether to enable hardware (DSR/DTR) flow control.
- **parity** (`Bytes()`) – Parity in the physical serial connection.
- **resource** (`Unicode()`) – Addressing information needed to make a connection to a device. Type and format are determined by the subclass implementation
- **rtscts** (`Bool()`) – Whether to enable hardware (RTS/CTS) flow control.
- **stopbits** (`Float(min=1,max=2,step=0.5)`) – Number of stop bits, one of *[1., 1.5, or 2.]*.
- **timeout** (`Float(min=0,max=inf)`) – Max time to wait for a connection before raising TimeoutError.
- **write_termination** (`Bytes()`) – Termination character to send after a write.
- **xonxoff** (`Bool()`) – Set *True* to enable software flow control.

connect()
Connect to the serial device with the VISA resource string defined in self.settings.resource

disconnect()
Disconnect the serial instrument

classmethod from_hwid(*hwid=None*, **args*, ***connection_params*)
Instantiate a new SerialDevice from a 'hwid' resource instead of a comport resource. A hwid string in windows might look something like:

r'PCIVEN_8086&DEV_9D3D&SUBSYS_06DC1028&REV_213&11583659&1&B3'

static list_ports(*hwid=None*)
List USB serial devices on the computer

Returns list of port resource information

class settings(*device*, **args*, ***kws*)
Bases: labbench.core.settings

Container for settings traits in a Device.

These settings are stored only on the host; setting or getting these values do not trigger live updates (or any communication) with the device. These define connection addressing information, communication settings, and options that only apply to implementing python support for the device.

The device uses this container to define the keyword options supported by its __init__ function. These are applied when you instantiate the device. After you instantiate the device, you can still change the setting with:

```
Device.settings.resource = 'insert-your-address-string-here'
```

trait attributes:

- *baud_rate*: *Int*
- *concurrency_support*: *Bool*
- *dsrdtr*: *Bool*
- *parity*: *Bytes*
- *resource*: *Unicode*
- *rtscts*: *Bool*
- *stopbits*: *Float*
- *timeout*: *Float*
- *write_termination*: *Bytes*
- *xonxoff*: *Bool*

baud_rate
Int(min=1)

Data rate of the physical serial connection.

concurrency_support
Bool(read_only=True)

Whether this backend supports threading

classmethod define(***kws*)
Change default values of the settings in parent settings, without redefining the full class. redefined according to each keyword argument. For example:

```
MyInstrumentClass.settings.define(parameter=7)
```

changes the default value of the *parameter* setting in *MyInstrumentClass.settings* to *7*. This is a convenience function to avoid completely redefining *parameter* if it was defined in a parent class of *MyInstrumentClass*.

dsrdtr
Bool()

Whether to enable hardware (DSR/DTR) flow control.

parity
Bytes()

Parity in the physical serial connection.

resource
Unicode()

Addressing information needed to make a connection to a device. Type and format are determined by the subclass implementation

rtscts
Bool()

Whether to enable hardware (RTS/CTS) flow control.

stopbits
Float(min=1,max=2,step=0.5)

Number of stop bits, one of *[1., 1.5, or 2.]*.

timeout
Float(min=0,max=inf)

Max time to wait for a connection before raising TimeoutError.

write_termination
Bytes()

Termination character to send after a write.

xonxoff
Bool()

Set *True* to enable software flow control.

class state(*device*, **args*, ***kws*)
Bases: `labbench.core.HasStateTraits`

Container for state traits in a Device. Getting or setting state traits triggers live updates: communication with the device to get or set the value on the Device. Therefore, getting or setting state traits needs the device to be connected.

To set a state value inside the device, use normal python assigment:

```
device.state.parameter = value
```

To get a state value from the device, you can also use it as a normal python variable:

```
variable = device.state.parameter + 1
```

trait attributes:

- *connected*: *Bool*

connected
Bool(read_only=True)

whether the `Device` instance is connected

classmethod getter(*func*)
Use this as a decorator to define a setter function for all traits in this class. The getter should take one argument: the instance of the trait to get. It should perform any operation needed to retrieve the current value of the device state corresponding to the supplied trait, using *self._device*.

One example is to send a command defined by trait.command.

The function should return a value that is the state from the device.

A trait that has its own getter defined will ignore this one.

classmethod setter(*func*)
Use this as a decorator to define a setter function for all traits in this class. The setter should take two arguments: the instance of the trait to get, and the value to set. It should perform any operation needed to apply the given value to the trait's state in *self._device*. One example is to send a command defined by trait.command.

Any return value from the function is ignored.

A trait that has its own setter defined will ignore this one.

class labbench.backends.**SerialLoggingDevice**(*resource=None*, ***settings*)
Bases: *labbench.backends.SerialDevice*

Manage connection, acquisition, and data retreival on a single GPS device. The goal is to make GPS devices controllable somewhat like instruments: maintaining their own threads, and blocking during setup or stop command execution.

Listener objects must implement an attach method with one argument consisting of the queue that the device manager uses to push data from the serial port.

Parameters

- **baud_rate** (`Int(min=1)`) – Data rate of the physical serial connection.
- **data_format** (`Bytes()`) – Data format metadata
- **dsrdtr** (`Bool()`) – Whether to enable hardware (DSR/DTR) flow control.
- **max_queue_size** (`Int(min=1)`) – Number of bytes to allocate in the data retreival buffer
- **parity** (`Bytes()`) – Parity in the physical serial connection.
- **poll_rate** (`Float(min=0,max=inf)`) – Data retreival rate from the device (in seconds)
- **resource** (`Unicode()`) – Addressing information needed to make a connection to a device. Type and format are determined by the subclass implementation
- **rtscts** (`Bool()`) – Whether to enable hardware (RTS/CTS) flow control.
- **stop_timeout** (`Float(min=0,max=inf)`) – Delay after a call to *stop* before terminating the runloop thread

- **stopbits** (Float *(min=1,max=2,step=0.5)*) – Number of stop bits, one of *[1., 1.5, or 2.]*.
- **timeout** (Float *(min=0,max=inf)*) – Max time to wait for a connection before raising TimeoutError.
- **write_termination** (Bytes *()*) – Termination character to send after a write.
- **xonxoff** (Bool *()*) – Set *True* to enable software flow control.

clear()
Throw away any log data in the buffer.

configure()
This is called at the beginning of the logging thread that runs on a call to *start*.

This is a stub that does nothing — it should be implemented by a subclass for a specific serial logger device.

connect()
Connect to the serial device with the VISA resource string defined in self.settings.resource

disconnect()
Disconnect the serial instrument

fetch()
Retrieve and return any log data in the buffer.

Returns any bytes in the buffer

classmethod from_hwid(*hwid=None*, **args*, ***connection_params*)
Instantiate a new SerialDevice from a 'hwid' resource instead of a comport resource. A hwid string in windows might look something like:

r'PCIVEN_8086&DEV_9D3D&SUBSYS_06DC1028&REV_213&11583659&1&B3'

static list_ports(*hwid=None*)
List USB serial devices on the computer

Returns list of port resource information

running()
Check whether the logger is running.

Returns *True* if the logger is running

class settings(*device*, **args*, ***kws*)
Bases: labbench.backends.settings

Container for settings traits in a Device.

These settings are stored only on the host; setting or getting these values do not trigger live updates (or any communication) with the device. These define connection addressing information, communication settings, and options that only apply to implementing python support for the device.

The device uses this container to define the keyword options supported by its __init__ function. These are applied when you instantiate the device. After you instantiate the device, you can still change the setting with:

```
Device.settings.resource = 'insert-your-address-string-here'
```

trait attributes:

- *baud_rate*: *Int*

- *concurrency_support*: *Bool*
- *data_format*: *Bytes*
- *dsrdtr*: *Bool*
- *max_queue_size*: *Int*
- *parity*: *Bytes*
- *poll_rate*: *Float*
- *resource*: *Unicode*
- *rtscts*: *Bool*
- *stop_timeout*: *Float*
- *stopbits*: *Float*
- *timeout*: *Float*
- *write_termination*: *Bytes*
- *xonxoff*: *Bool*

baud_rate
Int(min=1)

Data rate of the physical serial connection.

concurrency_support
Bool(read_only=True)

Whether this backend supports threading

data_format
Bytes()

Data format metadata

classmethod define(***kws*)
Change default values of the settings in parent settings, without redefining the full class. redefined according to each keyword argument. For example:

```
MyInstrumentClass.settings.define(parameter=7)
```

changes the default value of the *parameter* setting in *MyInstrumentClass.settings* to *7*. This is a convenience function to avoid completely redefining *parameter* if it was defined in a parent class of *MyInstrumentClass*.

dsrdtr
Bool()

Whether to enable hardware (DSR/DTR) flow control.

max_queue_size
Int(min=1)

Number of bytes to allocate in the data retreival buffer

parity
Bytes()

Parity in the physical serial connection.

poll_rate
Float(min=0,max=inf)

Data retreival rate from the device (in seconds)

resource
Unicode()

Addressing information needed to make a connection to a device. Type and format are determined by the subclass implementation

rtscts
Bool()

Whether to enable hardware (RTS/CTS) flow control.

stop_timeout
Float(min=0,max=inf)

Delay after a call to *stop* before terminating the runloop thread

stopbits
Float(min=1,max=2,step=0.5)

Number of stop bits, one of *[1., 1.5, or 2.]*.

timeout
Float(min=0,max=inf)

Max time to wait for a connection before raising TimeoutError.

write_termination
Bytes()

Termination character to send after a write.

xonxoff
Bool()

Set *True* to enable software flow control.

start()
Start a background thread that acquires log data into a queue.

Returns None

class state(*device*, **args*, ***kws*)
Bases: `labbench.core.HasStateTraits`

Container for state traits in a Device. Getting or setting state traits triggers live updates: communication with the device to get or set the value on the Device. Therefore, getting or setting state traits needs the device to be connected.

To set a state value inside the device, use normal python assigment:

```
device.state.parameter = value
```

To get a state value from the device, you can also use it as a normal python variable:

```
variable = device.state.parameter + 1
```

trait attributes:

- *connected*: *Bool*

connected
Bool(read_only=True)

whether the `Device` instance is connected

classmethod getter(*func*)
Use this as a decorator to define a setter function for all traits in this class. The getter should take one argument: the instance of the trait to get. It should perform any operation needed to retrieve the current value of the device state corresponding to the supplied trait, using *self._device*.

One example is to send a command defined by trait.command.

The function should return a value that is the state from the device.

A trait that has its own getter defined will ignore this one.

classmethod setter(*func*)
Use this as a decorator to define a setter function for all traits in this class. The setter should take two arguments: the instance of the trait to get, and the value to set. It should perform any operation needed to apply the given value to the trait's state in *self._device*. One example is to send a command defined by trait.command.

Any return value from the function is ignored.

A trait that has its own setter defined will ignore this one.

stop()
Stops the logger acquisition if it is running. Returns silently otherwise.

Returns None

class labbench.backends.**TelnetDevice**(*resource=None*, ***settings*)
Bases: *labbench.core.Device*

A general base class for communication devices via telnet. Unlike (for example) VISA instruments, there is no standardized command format like SCPI. The implementation is therefore limited to connect and disconnect, which open or close a pyserial connection object: the *backend* attribute. Subclasses can read or write with the backend attribute like they would any other telnetlib instance.

A TelnetDevice *resource* string is an IP address. The port is specified by *port*. These can be set when you instantiate the TelnetDevice or by setting them afterward in *settings*.

Subclassed devices that need state descriptors will need to implement *state.getter* and *state.setter* methods to implement the state set and get operations (as appropriate).

Parameters

- **port** (`Int(min=1)`) –
- **resource** (`Unicode()`) – Addressing information needed to make a connection to a device. Type and format are determined by the subclass implementation
- **timeout** (`Float(min=0,max=inf)`) – maximum time to wait for a connection before

connect()
Make the telnet connection to the host defined by the string in self.settings.resource

disconnect()
Disconnect the telnet connection

class settings(*device*, **args*, ***kws*)
Bases: `labbench.core.settings`

Container for settings traits in a Device.

These settings are stored only on the host; setting or getting these values do not trigger live updates (or any communication) with the device. These define connection addressing information, communication settings, and options that only apply to implementing python support for the device.

The device uses this container to define the keyword options supported by its __init__ function. These are applied when you instantiate the device. After you instantiate the device, you can still change the setting with:

```
Device.settings.resource = 'insert-your-address-string-here'
```

trait attributes:

- *concurrency_support*: *Bool*
- *port*: *Int*
- *resource*: *Unicode*
- *timeout*: *Float*

concurrency_support
Bool(read_only=True)

Whether this backend supports threading

classmethod define(***kws*)
Change default values of the settings in parent settings, without redefining the full class. redefined according to each keyword argument. For example:

```
MyInstrumentClass.settings.define(parameter=7)
```

changes the default value of the *parameter* setting in *MyInstrumentClass.settings* to *7*. This is a convenience function to avoid completely redefining *parameter* if it was defined in a parent class of *MyInstrumentClass*.

port
Int(min=1)

resource
Unicode()

Addressing information needed to make a connection to a device. Type and format are determined by the subclass implementation

timeout
Float(min=0,max=inf)

maximum time to wait for a connection before

class state(*device*, **args*, ***kws*)
Bases: `labbench.core.HasStateTraits`

Container for state traits in a Device. Getting or setting state traits triggers live updates: communication with the device to get or set the value on the Device. Therefore, getting or setting state traits needs the device to be connected.

To set a state value inside the device, use normal python assigment:

```
device.state.parameter = value
```

To get a state value from the device, you can also use it as a normal python variable:

```
variable = device.state.parameter + 1
```

trait attributes:

- *connected*: *Bool*

connected
Bool(read_only=True)

whether the `Device` instance is connected

classmethod getter(*func*)
Use this as a decorator to define a setter function for all traits in this class. The getter should take one argument: the instance of the trait to get. It should perform any operation needed to retrieve the current value of the device state corresponding to the supplied trait, using *self._device*.

One example is to send a command defined by trait.command.

The function should return a value that is the state from the device.

A trait that has its own getter defined will ignore this one.

classmethod setter(*func*)
Use this as a decorator to define a setter function for all traits in this class. The setter should take two arguments: the instance of the trait to get, and the value to set. It should perform any operation needed to apply the given value to the trait's state in *self._device*. One example is to send a command defined by trait.command.

Any return value from the function is ignored.

A trait that has its own setter defined will ignore this one.

class labbench.backends.**VISADevice**(*resource=None*, ***settings*)
Bases: *labbench.core.Device*

class VISADevice(*resource*, *read_termination='\n'*, *write_termination='\n'*)

VISADevice instances control VISA instruments using a pyvisa backend. Compared to direct use of pyvisa, this style of use permits use of labbench device *state* goodies for compact, readable code, as well as type checking.

For example, the following fetches the identity string from the remote instrument:

```
with VISADevice('USB0::0x2A8D::0x1E01::SG56360004::INSTR') as instr:
    print inst.state.identity
```

This is equivalent to the more pyvisa-style use as follows:

```
inst = VISADevice('USB0::0x2A8D::0x1E01::SG56360004::INSTR')
inst.connect()
print inst.query('*IDN?')
```

Use of *inst.state* makes it possible to add callbacks to support automatic state logging, or to build a UI.

Parameters **resource** (*Unicode()*) – Addressing information needed to make a connection to a device. Type and format are determined by the subclass implementation

connect()
Connect to the VISA instrument defined by the VISA resource set by *self.settings.resource*. The pyvisa backend object is assigned to *self.backend*.

Returns None

Instead of calling *connect* directly, consider using *with* statements to guarantee proper disconnection if there is an error. For example, the following sets up a connected instance:

```
with VISADevice('USB0::0x2A8D::0x1E01::SG56360004::INSTR') as inst:
    print inst.state.identity
    print inst.state.status_byte
    print inst.state.options
```

would instantiate a *VISADevice* and guarantee it is disconnected either at the successful completion of the *with* block, or if there is any exception.

disconnect()
Disconnect the VISA instrument. If you use a *with* block this is handled automatically and you do not need to call this method.

Returns None

classmethod list_resources()
List the resource strings of the available devices sensed by the VISA backend.

overlap_and_block(*timeout=None*, *quiet=False*)
A request is sent to the instrument to overlap all of the VISA commands written while in this context. At the end of the block, wait until the instrument confirms that all operations have finished. This is the standard VISA ';*OPC' and '*OPC?' behavior.

This is meant to be used in *with* blocks as follows:

```
with inst.overlap_and_block():
    inst.write('long running command 1')
    inst.write('long running command 2')
```

The wait happens on leaving the *with* block.

Parameters

- **timeout** – delay (in milliseconds) on waiting for the instrument to finish the overlapped commands before a TimeoutError after leaving the *with* block. If *None*, use self.backend.timeout.
- **quiet** – Suppress timeout exceptions if this evaluates as True

preset()
Convenience function to send standard SCPI '*RST'

query(*msg*, *timeout=None*)
Query an SCPI command to the device with pyvisa, and return a string containing the device response.

Handles debug logging and adjustments when in overlap_and_block contexts as appropriate.

Parameters **msg** (*str*) – the SCPI command to send by VISA

Returns the response to the query from the device

classmethod set_backend(*backend_name*)
Set the pyvisa resource manager for all VISA objects.

Parameters **str** (*backend_name*) – '@ni' (the default) or '@py'

Returns None

class settings(*device*, **args*, ***kws*)
Bases: labbench.core.settings

Container for settings traits in a Device.

These settings are stored only on the host; setting or getting these values do not trigger live updates (or any communication) with the device. These define connection addressing information, communication settings, and options that only apply to implementing python support for the device.

The device uses this container to define the keyword options supported by its __init__ function. These are applied when you instantiate the device. After you instantiate the device, you can still change the setting with:

```
Device.settings.resource = 'insert-your-address-string-here'
```

trait attributes:

- *concurrency_support*: *Bool*
- *read_termination*: *Unicode*
- *resource*: *Unicode*
- *write_termination*: *Unicode*

concurrency_support
Bool(read_only=True)

Whether this backend supports threading

classmethod define(***kws*)
Change default values of the settings in parent settings, without redefining the full class. redefined according to each keyword argument. For example:

```
MyInstrumentClass.settings.define(parameter=7)
```

changes the default value of the *parameter* setting in *MyInstrumentClass.settings* to *7*. This is a convenience function to avoid completely redefining *parameter* if it was defined in a parent class of *MyInstrumentClass*.

read_termination
Unicode(read_only='connected')

termination character to indicate end of message on receive from the instrument

resource
Unicode()

Addressing information needed to make a connection to a device. Type and format are determined by the subclass implementation

write_termination
Unicode(read_only='connected')

termination character to indicate end of message in messages sent to the instrument

class state(*device*, **args*, ***kws*)
Bases: `labbench.core.state`

Container for state traits in a Device. Getting or setting state traits triggers live updates: communication with the device to get or set the value on the Device. Therefore, getting or setting state traits needs the device to be connected.

To set a state value inside the device, use normal python assigment:

```
device.state.parameter = value
```

To get a state value from the device, you can also use it as a normal python variable:

```
variable = device.state.parameter + 1
```

trait attributes:

- *connected*: *Bool*
- *identity*: *Unicode*
- *options*: *Unicode*
- *status_byte*: *Dict*

connected
Bool(read_only=True)

whether the `Device` instance is connected

classmethod getter (*func*)
Use this as a decorator to define a setter function for all traits in this class. The getter should take one argument: the instance of the trait to get. It should perform any operation needed to retrieve the current value of the device state corresponding to the supplied trait, using *self._device*.

One example is to send a command defined by trait.command.

The function should return a value that is the state from the device.

A trait that has its own getter defined will ignore this one.

identity
Unicode(command='*IDN',read_only=True,cache=True)

identity string reported by the instrument

options
Unicode(command='*OPT',read_only=True,cache=True)

options reported by the instrument

classmethod setter (*func*)
Use this as a decorator to define a setter function for all traits in this class. The setter should take two arguments: the instance of the trait to get, and the value to set. It should perform any operation needed to apply the given value to the trait's state in *self._device*. One example is to send a command defined by trait.command.

Any return value from the function is ignored.

A trait that has its own setter defined will ignore this one.

status_byte
Dict(command='*STB',read_only=True)

VISA status byte reported by the instrument

class suppress_timeout (**exceptions*)
Bases: `contextlib.suppress`

Context manager to suppress timeout exceptions.

Example:

```
with inst.suppress_timeout():
    inst.write('long running command 1')
    inst.write('long running command 2')
```

If the command 1 raises an exception, then command 2 will (silently) not execute.

wait()
Convenience function to send standard SCPI '*WAI'

write(*msg*)
Write an SCPI command to the device with pyvisa.

Handles debug logging and adjustments when in overlap_and_block contexts as appropriate.

Parameters **msg** (*str*) – the SCPI command to send by VISA

Returns None

class labbench.backends.**Win32ComDevice**(*resource=None*, ***settings*)
Bases: *labbench.core.Device*

Basic support for calling win32 COM APIs.

The python wrappers for COM drivers still basically require that threading is performed using the windows COM API, and not the python threading. Figuring this out with win32com calls within python is not for the faint of heart. Threading support is instead realized with util.ThreadSandbox, which ensures that all calls to the dispatched COM object block until the previous calls are completed from within a background thread. Set concurrency_support=True to decide whether this thread support wrapper is applied to the dispatched Win32Com object.

Parameters

- **com_object** (Unicode *()*) – the win32com object string
- **concurrency_support** (Bool *()*) – whether this Device implementation supports threading
- **resource** (Unicode *()*) – Addressing information needed to make a connection to a device. Type and format are determined by the subclass implementation

connect()
Connect to the win32 com object

disconnect()
Backend implementations must overload this to disconnect an existing connection to the resource encapsulated in the object.

class settings(*device*, **args*, ***kws*)
Bases: labbench.core.settings

Container for settings traits in a Device.

These settings are stored only on the host; setting or getting these values do not trigger live updates (or any communication) with the device. These define connection addressing information, communication settings, and options that only apply to implementing python support for the device.

The device uses this container to define the keyword options supported by its __init__ function. These are applied when you instantiate the device. After you instantiate the device, you can still change the setting with:

```
Device.settings.resource = 'insert-your-address-string-here'
```

trait attributes:

- *com_object*: *Unicode*
- *concurrency_support*: *Bool*

- *resource*: *Unicode*

com_object
Unicode()

the win32com object string

concurrency_support
Bool()

whether this `Device` implementation supports threading

classmethod define(***kws*)
Change default values of the settings in parent settings, without redefining the full class. redefined according to each keyword argument. For example:

```
MyInstrumentClass.settings.define(parameter=7)
```

changes the default value of the *parameter* setting in *MyInstrumentClass.settings* to *7*. This is a convenience function to avoid completely redefining *parameter* if it was defined in a parent class of *MyInstrumentClass*.

resource
Unicode()

Addressing information needed to make a connection to a device. Type and format are determined by the subclass implementation

class state(*device*, **args*, ***kws*)
Bases: `labbench.core.HasStateTraits`

Container for state traits in a Device. Getting or setting state traits triggers live updates: communication with the device to get or set the value on the Device. Therefore, getting or setting state traits needs the device to be connected.

To set a state value inside the device, use normal python assigment:

```
device.state.parameter = value
```

To get a state value from the device, you can also use it as a normal python variable:

```
variable = device.state.parameter + 1
```

trait attributes:

- *connected*: *Bool*

connected
Bool(read_only=True)

whether the `Device` instance is connected

classmethod getter(*func*)
Use this as a decorator to define a setter function for all traits in this class. The getter should take one argument: the instance of the trait to get. It should perform any operation needed to retrieve the current value of the device state corresponding to the supplied trait, using *self._device*.

One example is to send a command defined by trait.command.

The function should return a value that is the state from the device.

A trait that has its own getter defined will ignore this one.

classmethod setter(*func*)

Use this as a decorator to define a setter function for all traits in this class. The setter should take two arguments: the instance of the trait to get, and the value to set. It should perform any operation needed to apply the given value to the trait's state in *self._device*. One example is to send a command defined by trait.command.

Any return value from the function is ignored.

A trait that has its own setter defined will ignore this one.

CHAPTER 2

labbench.core module

This implementation is deeply intertwined with low-level internals of traitlets and obscure details of the python object model. Consider reading the documentation closely and inheriting these objects instead of reverse-engineering this code.

exception labbench.core.**ConnectionError**
Bases: traitlets.traitlets.TraitError

Failure on attempt to connect to a device

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception labbench.core.**DeviceException**
Bases: Exception

Generic Device exception

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception labbench.core.**DeviceNotReady**
Bases: Exception

Failure to communicate with the Device because it was not ready for communication

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception labbench.core.**DeviceFatalError**
Bases: Exception

A fatal error in the device

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception labbench.core.**DeviceConnectionLost**
Bases: Exception

Connection state has been lost unexpectedly

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception labbench.core.**DeviceStateError**
Bases: traitlets.traitlets.TraitError

Failure to get or set a state in *Device.state*

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class labbench.core.**Int**(*default_value=traitlets.Undefined*, *allow_none=False*, *read_only=None*, *help=None*, *write_only=None*, *cache=None*, *command=None*, *getter=None*, *setter=None*, *remap={}*, ***kwargs*)
Bases: labbench.core.TraitMixIn, traitlets.traitlets.CInt

Trait for an integer value, with type and bounds checking.

Parameters

- **default_value** – initial value (in *settings* only, not *state*)
- **allow_none** – whether to allow pythonic *None* to represent a null value
- **read_only** – True if this should not accept a set (write) operation
- **write_only** – True if this should not accept a get (read) operation (in *state* only, not *settings*)
- **cache** – True if this should only read from the device once, then return that value in future calls (in *state* only, not *settings*)
- **getter** – Function or other callable (no arguments) that retrieves the value from the remote device, or None (in *state* only, not *settings*)
- **setter** – Function or other callable (one *value* argument) that sets the value from the remote device, or None (in *state* only, not *settings*)
- **remap** – A dictionary {python_value: device_representation} to use as a look-up table that transforms python representation into the format expected by a device
- **min** – lower bound for the value
- **max** – upper bound for the value

class labbench.core.**Float**(*default_value=traitlets.Undefined*, *allow_none=False*, *read_only=None*, *help=None*, *write_only=None*, *cache=None*, *command=None*, *getter=None*, *setter=None*, *remap={}*, ***kwargs*)
Bases: labbench.core.TraitMixIn, labbench.core.CFLoatSteppedTraitlet

Trait for a floating point value, with type and bounds checking.

Parameters

- **default_value** – initial value (in *settings* only, not *state*)
- **allow_none** – whether to allow pythonic *None* to represent a null value
- **read_only** – True if this should not accept a set (write) operation
- **write_only** – True if this should not accept a get (read) operation (in *state* only, not *settings*)
- **cache** – True if this should only read from the device once, then return that value in future calls (in *state* only, not *settings*)

- **getter** – Function or other callable (no arguments) that retrieves the value from the remote device, or None (in *state* only, not *settings*)
- **setter** – Function or other callable (one *value* argument) that sets the value from the remote device, or None (in *state* only, not *settings*)
- **remap** – A dictionary {python_value: device_representation} to use as a look-up table that transforms python representation into the format expected by a device
- **min** – lower bound for the value
- **max** – upper bound for the value

class labbench.core.**Unicode**(*default_value=traitlets.Undefined*, *allow_none=False*, *read_only=None*, *help=None*, *write_only=None*, *cache=None*, *command=None*, *getter=None*, *setter=None*, *remap={}*, ***kwargs*)

Bases: labbench.core.TraitMixIn, traitlets.traitlets.CUnicode

Trait for a Unicode string value, with type checking.

Parameters

- **default_value** – initial value (in *settings* only, not *state*)
- **allow_none** – whether to allow pythonic *None* to represent a null value
- **read_only** – True if this should not accept a set (write) operation
- **write_only** – True if this should not accept a get (read) operation (in *state* only, not *settings*)
- **cache** – True if this should only read from the device once, then return that value in future calls (in *state* only, not *settings*)
- **getter** – Function or other callable (no arguments) that retrieves the value from the remote device, or None (in *state* only, not *settings*)
- **setter** – Function or other callable (one *value* argument) that sets the value from the remote device, or None (in *state* only, not *settings*)
- **remap** – A dictionary {python_value: device_representation} to use as a look-up table that transforms python representation into the format expected by a device

class labbench.core.**Complex**(*default_value=traitlets.Undefined*, *allow_none=False*, *read_only=None*, *help=None*, *write_only=None*, *cache=None*, *command=None*, *getter=None*, *setter=None*, *remap={}*, ***kwargs*)

Bases: labbench.core.TraitMixIn, traitlets.traitlets.CComplex

Trait for a complex numeric value, with type checking.

Parameters

- **default_value** – initial value (in *settings* only, not *state*)
- **allow_none** – whether to allow pythonic *None* to represent a null value
- **read_only** – True if this should not accept a set (write) operation
- **write_only** – True if this should not accept a get (read) operation (in *state* only, not *settings*)
- **cache** – True if this should only read from the device once, then return that value in future calls (in *state* only, not *settings*)
- **getter** – Function or other callable (no arguments) that retrieves the value from the remote device, or None (in *state* only, not *settings*)

- **setter** – Function or other callable (one *value* argument) that sets the value from the remote device, or None (in *state* only, not *settings*)
- **remap** – A dictionary {python_value: device_representation} to use as a look-up table that transforms python representation into the format expected by a device

class labbench.core.**Bytes**(*default_value=traitlets.Undefined*, *allow_none=False*, *read_only=None*, *help=None*, *write_only=None*, *cache=None*, *command=None*, *getter=None*, *setter=None*, *remap={}*, ***kwargs*)

Bases: labbench.core.TraitMixIn, traitlets.traitlets.CBytes

Trait for a byte string value, with type checking.

Parameters

- **default_value** – initial value (in *settings* only, not *state*)
- **allow_none** – whether to allow pythonic *None* to represent a null value
- **read_only** – True if this should not accept a set (write) operation
- **write_only** – True if this should not accept a get (read) operation (in *state* only, not *settings*)
- **cache** – True if this should only read from the device once, then return that value in future calls (in *state* only, not *settings*)
- **getter** – Function or other callable (no arguments) that retrieves the value from the remote device, or None (in *state* only, not *settings*)
- **setter** – Function or other callable (one *value* argument) that sets the value from the remote device, or None (in *state* only, not *settings*)
- **remap** – A dictionary {python_value: device_representation} to use as a look-up table that transforms python representation into the format expected by a device

class labbench.core.**CaselessBytesEnum**(*default_value=traitlets.Undefined*, *allow_none=False*, *read_only=None*, *help=None*, *write_only=None*, *cache=None*, *command=None*, *getter=None*, *setter=None*, *remap={}*, ***kwargs*)

Bases: labbench.core.TraitMixIn, labbench.core.EnumBytesTraitlet

Trait for an enumerated list of valid case-insensitive byte string values, with type checking.

Parameters

- **default_value** – initial value (in *settings* only, not *state*)
- **allow_none** – whether to allow pythonic *None* to represent a null value
- **read_only** – True if this should not accept a set (write) operation
- **write_only** – True if this should not accept a get (read) operation (in *state* only, not *settings*)
- **cache** – True if this should only read from the device once, then return that value in future calls (in *state* only, not *settings*)
- **getter** – Function or other callable (no arguments) that retrieves the value from the remote device, or None (in *state* only, not *settings*)
- **setter** – Function or other callable (one *value* argument) that sets the value from the remote device, or None (in *state* only, not *settings*)
- **remap** – A dictionary {python_value: device_representation} to use as a look-up table that transforms python representation into the format expected by a device

- **values** – An iterable of valid byte strings to accept
- **case_sensitive** – Whether to be case_sensitive

class labbench.core.**Bool**(*default_value=traitlets.Undefined*, *allow_none=False*, *read_only=None*, *help=None*, *write_only=None*, *cache=None*, *command=None*, *getter=None*, *setter=None*, *remap={}*, ***kwargs*)

Bases: labbench.core.TraitMixIn, traitlets.traitlets.CBool

Trait for a python boolean, with type checking.

Parameters

- **default_value** – initial value (in *settings* only, not *state*)
- **allow_none** – whether to allow pythonic *None* to represent a null value
- **read_only** – True if this should not accept a set (write) operation
- **write_only** – True if this should not accept a get (read) operation (in *state* only, not *settings*)
- **cache** – True if this should only read from the device once, then return that value in future calls (in *state* only, not *settings*)
- **getter** – Function or other callable (no arguments) that retrieves the value from the remote device, or None (in *state* only, not *settings*)
- **setter** – Function or other callable (one *value* argument) that sets the value from the remote device, or None (in *state* only, not *settings*)
- **remap** – A dictionary {python_value: device_representation} to use as a look-up table that transforms python representation into the format expected by a device

class labbench.core.**List**(*default_value=traitlets.Undefined*, *allow_none=False*, *read_only=None*, *help=None*, *write_only=None*, *cache=None*, *command=None*, *getter=None*, *setter=None*, *remap={}*, ***kwargs*)

Bases: labbench.core.TraitMixIn, traitlets.traitlets.List

Trait for a python list value, with type checking.

Parameters

- **default_value** – initial value (in *settings* only, not *state*)
- **allow_none** – whether to allow pythonic *None* to represent a null value
- **read_only** – True if this should not accept a set (write) operation
- **write_only** – True if this should not accept a get (read) operation (in *state* only, not *settings*)
- **cache** – True if this should only read from the device once, then return that value in future calls (in *state* only, not *settings*)
- **getter** – Function or other callable (no arguments) that retrieves the value from the remote device, or None (in *state* only, not *settings*)
- **setter** – Function or other callable (one *value* argument) that sets the value from the remote device, or None (in *state* only, not *settings*)
- **remap** – A dictionary {python_value: device_representation} to use as a look-up table that transforms python representation into the format expected by a device

class_init(*cls*, *name*)

Part of the initialization which may depend on the underlying HasDescriptors class.

It is typically overloaded for specific types.

This method is called by `MetaHasDescriptors.__init__()` passing the class (*cls*) and *name* under which the descriptor has been assigned.

instance_init(*obj*)
Part of the initialization which may depend on the underlying HasDescriptors instance.

It is typically overloaded for specific types.

This method is called by `HasTraits.__new__()` and in the `BaseDescriptor.instance_init()` method of descriptors holding other descriptors.

klass
alias of `builtins.list`

class labbench.core.**Dict**(*default_value=traitlets.Undefined*, *allow_none=False*, *read_only=None*, *help=None*, *write_only=None*, *cache=None*, *command=None*, *getter=None*, *setter=None*, *remap={}*, ***kwargs*)
Bases: `labbench.core.TraitMixIn`, `traitlets.traitlets.Dict`

Trait for a python dict value, with type checking.

Parameters

- **default_value** – initial value (in *settings* only, not *state*)
- **allow_none** – whether to allow pythonic *None* to represent a null value
- **read_only** – True if this should not accept a set (write) operation
- **write_only** – True if this should not accept a get (read) operation (in *state* only, not *settings*)
- **cache** – True if this should only read from the device once, then return that value in future calls (in *state* only, not *settings*)
- **getter** – Function or other callable (no arguments) that retrieves the value from the remote device, or None (in *state* only, not *settings*)
- **setter** – Function or other callable (one *value* argument) that sets the value from the remote device, or None (in *state* only, not *settings*)
- **remap** – A dictionary {python_value: device_representation} to use as a look-up table that transforms python representation into the format expected by a device

class_init(*cls*, *name*)
Part of the initialization which may depend on the underlying HasDescriptors class.

It is typically overloaded for specific types.

This method is called by `MetaHasDescriptors.__init__()` passing the class (*cls*) and *name* under which the descriptor has been assigned.

instance_init(*obj*)
Part of the initialization which may depend on the underlying HasDescriptors instance.

It is typically overloaded for specific types.

This method is called by `HasTraits.__new__()` and in the `BaseDescriptor.instance_init()` method of descriptors holding other descriptors.

class labbench.core.**TCPAddress**(*default_value=traitlets.Undefined*, *allow_none=False*, *read_only=None*, *help=None*, *write_only=None*, *cache=None*, *command=None*, *getter=None*, *setter=None*, *remap={}*, ***kwargs*)

Bases: labbench.core.TraitMixIn, traitlets.traitlets.TCPAddress

Trait for a (address, port) TCP address tuple value, with type checking.

Parameters

- **default_value** – initial value (in *settings* only, not *state*)
- **allow_none** – whether to allow pythonic *None* to represent a null value
- **read_only** – True if this should not accept a set (write) operation
- **write_only** – True if this should not accept a get (read) operation (in *state* only, not *settings*)
- **cache** – True if this should only read from the device once, then return that value in future calls (in *state* only, not *settings*)
- **getter** – Function or other callable (no arguments) that retrieves the value from the remote device, or None (in *state* only, not *settings*)
- **setter** – Function or other callable (one *value* argument) that sets the value from the remote device, or None (in *state* only, not *settings*)
- **remap** – A dictionary {python_value: device_representation} to use as a look-up table that transforms python representation into the format expected by a device

class labbench.core.**CaselessStrEnum**(*default_value=traitlets.Undefined*, *allow_none=False*, *read_only=None*, *help=None*, *write_only=None*, *cache=None*, *command=None*, *getter=None*, *setter=None*, *remap={}*, ***kwargs*)

Bases: labbench.core.TraitMixIn, traitlets.traitlets.CaselessStrEnum

Trait for an enumerated list of valid case-insensitive unicode string values, with type checking.

Parameters

- **default_value** – initial value (in *settings* only, not *state*)
- **allow_none** – whether to allow pythonic *None* to represent a null value
- **read_only** – True if this should not accept a set (write) operation
- **write_only** – True if this should not accept a get (read) operation (in *state* only, not *settings*)
- **cache** – True if this should only read from the device once, then return that value in future calls (in *state* only, not *settings*)
- **getter** – Function or other callable (no arguments) that retrieves the value from the remote device, or None (in *state* only, not *settings*)
- **setter** – Function or other callable (one *value* argument) that sets the value from the remote device, or None (in *state* only, not *settings*)
- **remap** – A dictionary {python_value: device_representation} to use as a look-up table that transforms python representation into the format expected by a device
- **values** – An iterable of valid unicode strings to accept

info()

Returns a description of the trait.

class labbench.core.**Device**(*resource=None*, ***settings*)
Bases: object

Device is the base class common to all labbench drivers. Inherit it to implement a backend, or a specialized type of driver.

Drivers that subclass *Device* get

- device connection management via context management (the *with* statement)
- test state management for easy test logging and extension to UI
- a degree automatic stylistic consistency between drivers

Parameters

- **resource** – resource identifier, with type and format determined by backend (see specific subclasses for details)
- ****local_states** – set the local state for each supplied state key and value

---

Note: Use *Device* by subclassing it only if you are implementing a driver that needs a new type of backend.

Several types of backends have already been implemented as part of labbench:

- VISADevice exposes a pyvisa backend for VISA Instruments
- CommandLineWrapper exposes a threaded pipes backend for command line tools
- Serial exposes a pyserial backend for serial port communication
- DotNetDevice exposes a pythonnet for wrapping dotnet libraries

(and others). If you are implementing a driver that uses one of these backends, inherit from the corresponding class above, not *Device*.

Parameters **resource** (Unicode *()*) – Addressing information needed to make a connection to a device. Type and format are determined by the subclass implementation

backend = DisconnectedBackend()
it is to be set in *connect* and *disconnect* by the subclass that implements the backend.

connect()
Backend implementations overload this to open a backend connection to the resource.

disconnect()
Backend implementations must overload this to disconnect an existing connection to the resource encapsulated in the object.

class settings(*device*, **args*, ***kws*)
Bases: labbench.core.HasSettingsTraits

Container for settings traits in a Device.

These settings are stored only on the host; setting or getting these values do not trigger live updates (or any communication) with the device. These define connection addressing information, communication settings, and options that only apply to implementing python support for the device.

The device uses this container to define the keyword options supported by its __init__ function. These are applied when you instantiate the device. After you instantiate the device, you can still change the setting with:

```
Device.settings.resource = 'insert-your-address-string-here'
```

trait attributes:

- *concurrency_support*: *Bool*
- *resource*: *Unicode*

concurrency_support
Bool(read_only=True)

Whether this backend supports threading

classmethod define(***kws*)
Change default values of the settings in parent settings, without redefining the full class. redefined according to each keyword argument. For example:

```
MyInstrumentClass.settings.define(parameter=7)
```

changes the default value of the *parameter* setting in *MyInstrumentClass.settings* to *7*. This is a convenience function to avoid completely redefining *parameter* if it was defined in a parent class of *MyInstrumentClass*.

resource
Unicode()

Addressing information needed to make a connection to a device. Type and format are determined by the subclass implementation

class state(*device*, **args*, ***kws*)
Bases: `labbench.core.HasStateTraits`

Container for state traits in a Device. Getting or setting state traits triggers live updates: communication with the device to get or set the value on the Device. Therefore, getting or setting state traits needs the device to be connected.

To set a state value inside the device, use normal python assigment:

```
device.state.parameter = value
```

To get a state value from the device, you can also use it as a normal python variable:

```
variable = device.state.parameter + 1
```

trait attributes:

- *connected*: *Bool*

connected
Bool(read_only=True)

whether the *Device* instance is connected

classmethod getter(*func*)
Use this as a decorator to define a setter function for all traits in this class. The getter should take one argument: the instance of the trait to get. It should perform any operation needed to retrieve the current value of the device state corresponding to the supplied trait, using *self._device*.

One example is to send a command defined by trait.command.

The function should return a value that is the state from the device.

A trait that has its own getter defined will ignore this one.

classmethod setter(*func*)
Use this as a decorator to define a setter function for all traits in this class. The setter should take two arguments: the instance of the trait to get, and the value to set. It should perform any operation needed to apply the given value to the trait's state in *self._device*. One example is to send a command defined by trait.command.

Any return value from the function is ignored.

A trait that has its own setter defined will ignore this one.

labbench.core.**list_devices**(*depth=1*)
Look for Device instances, and their names, in the calling code context (depth == 1) or its callers (if depth in (2,3,...)). Checks locals() in that context first. If no Device instances are found there, search the first argument of the first function argument, in case this is a method in a class.

exception labbench.core.**CommandNotImplementedError**
Bases: NotImplementedError

A command that has been defined but not implemented

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

CHAPTER 3

labbench.data module

class labbench.data.**StateAggregator**
Bases: object

Aggregate state information from multiple devices. This can be the basis for automatic database logging.

get()
Aggregate and return the current device states as configured with *observe()*.

Returns dictionary of aggregated states. Keys are strings defined by *key()* (defaults to '{device name}_{state name}'). Values are the type and value of the corresponding state of the device instance.

key(*device_name*, *state_name*)
Generate a name for a state based on the names of a device and one of its states or settings.

observe(*devices*, *changes=True*, *always=[]*, *never=[]*)
Deprecated - use *observe_states* instead

observe_settings(*devices, changes=True, always=[], never=['connected']*)
Configure Each time a device setting is set from python, intercept the value to include in the aggregate state.

Device may be a single device instance, or an several devices in an iterable (such as a list or tuple) to apply to each one.

Subsequent calls to *observe_settings()* replace the existing list of observed states for each device.

Parameters

- **devices** – Device instance or iterable of Device instances
- **changes** (*bool*) – Whether to automatically log each time a state is set for the supplied device(s)
- **always** – name (or iterable of multiple names) of settings to actively update on each call to get()
- **never** – name (or iterable of multiple names) of settings to exclude from aggregated result (overrides **:param:'always'**)

observe_states (*devices, changes=True, always=[], never=['connected']*)
Configure Each time a device state is set from python, intercept the value to include in the aggregate state.

Device may be a single device instance, or an several devices in an iterable (such as a list or tuple) to apply to each one.

Subsequent calls to *observe_states()* replace the existing list of observed states for each device.

Parameters

- **devices** – Device instance or iterable of Device instances
- **changes** (*bool*) – Whether to automatically log each time a state is set for the supplied device(s)
- **always** – name (or iterable of multiple names) of states to actively update on each call to get()
- **never** – name (or iterable of multiple names) of states to exclude from aggregated result (overrides **:param:'always'**)

set_device_labels (***mapping*)
Manually choose device name for a device instance.

Parameters **mapping** (*dict*) – name mapping, formatted as {device_object: 'device name'}

Returns None

class labbench.data.**StatesToRelationalTable**(*path, overwrite=False, text_relational_min=1024, force_relational=['host_log'], dirname_fmt='{id} {host_time}', nonscalar_file_type='csv', metadata_dirname='metadata', tar=False, **metadata*)

Bases: *labbench.data.StateAggregator*

Abstract base class for loggers that queue dictionaries of data before writing to disk. This extends *StateAggregator* to support

1. queuing aggregate state of devices by lists of dictionaries;
2. custom metadata in each queued aggregate state entry; and
3. custom response to non-scalar data (such as relational databasing).

Parameters

- **path** (*str*) – Base path to use for the master database
- **overwrite** (*bool*) – Whether to overwrite the master database if it exists (otherwise, append)
- **text_relational_min** – Text with at least this many characters is stored as a relational text file instead of directly in the database
- **force_relational** – A list of columns that should always be stored as relational data instead of directly in the database
- **nonscalar_file_type** – The data type to use in non-scalar (tabular, vector, etc.) relational data
- **metadata_dirname** – The name of the subdirectory that should be used to store metadata (device connection parameters, etc.)

- **tar** – Whether to store the relational data within directories in a tar file, instead of subdirectories

append(**args*, ***kwargs*)
Add a new row of data to the list of data that awaits write to disk.

This cache of pending data row is in the dictionary *self.pending*. Each row is represented as a dictionary of pairs formatted as {'column_name': 'row_value'}. These pairs come from a combination of 1) keyword arguments passed as *kwargs*, 2) a single dictionary argument, and/or 3) state traits configured automatically with *self.observe_states*.

The first pass at forming the row is the single dictionary argument

```
row = {'name1': value1, 'name2': value2, 'name3': value3}
db.append(row)
```

The second pass is to update with values as configured with *self.observe_states*.

Keyword arguments are passed in as

```
db.append(name1=value1, name2=value2, nameN=valueN)
```

Simple "scalar" database types like numbers, booleans, and strings are added directly to the table. Non-scalar or multidimensional values are stored in a separate file (as defined in *set_path_format()*), and the path to this file is stored in the table.

The row of data is appended to list of rows pending write to disk, *self.pending*. Nothing is written to disk until *write()*.

Parameters **copy=True** (*bool*) – When *True* (the default), use a deep copy of *data* to avoid problems with overwriting references to data if *data* is reused during test. This takes some extra time; set to *False* to skip this copy operation.

Returns the dictionary representation of the row added to *self.pending*.

clear()
Remove any queued data that has been added by append.

close()
Close the file or database connection. This is an abstract base method (to be overridden by inheriting classes)

Returns None

get()
Aggregate and return the current device states as configured with *observe()*.

Returns dictionary of aggregated states. Keys are strings defined by *key()* (defaults to '{device name}_{state name}'). Values are the type and value of the corresponding state of the device instance.

key(*device_name*, *state_name*)
Generate a name for a state based on the names of a device and one of its states or settings.

observe(*devices*, *changes=True*, *always=[]*, *never=[]*)
Deprecated - use *observe_states* instead

observe_settings(*devices, changes=True, always=[], never=['connected']*)
Configure Each time a device setting is set from python, intercept the value to include in the aggregate state.

Device may be a single device instance, or an several devices in an iterable (such as a list or tuple) to apply to each one.

Subsequent calls to *observe_settings()* replace the existing list of observed states for each device.

Parameters

- **devices** – Device instance or iterable of Device instances
- **changes** (*bool*) – Whether to automatically log each time a state is set for the supplied device(s)
- **always** – name (or iterable of multiple names) of settings to actively update on each call to get()
- **never** – name (or iterable of multiple names) of settings to exclude from aggregated result (overrides **:param:'always'**)

observe_states (*devices, changes=True, always=[], never=['connected']*)

Configure Each time a device state is set from python, intercept the value to include in the aggregate state.

Device may be a single device instance, or an several devices in an iterable (such as a list or tuple) to apply to each one.

Subsequent calls to *observe_states()* replace the existing list of observed states for each device.

Parameters

- **devices** – Device instance or iterable of Device instances
- **changes** (*bool*) – Whether to automatically log each time a state is set for the supplied device(s)
- **always** – name (or iterable of multiple names) of states to actively update on each call to get()
- **never** – name (or iterable of multiple names) of states to exclude from aggregated result (overrides **:param:'always'**)

open (*path=None*)

This must be implemented by a subclass to open the data storage resource.

set_device_labels (***mapping*)

Manually choose device name for a device instance.

Parameters **mapping** (*dict*) – name mapping, formatted as {device_object: 'device name'}

Returns None

set_path_format (*format*)

Set the path name convention for relational files that is used when a table entry contains non-scalar (multidimensional) information and will need to be stored in a separate file. The entry in the aggregate states table becomes the path to the file.

The format string follows the syntax of python's python's built-in `str.format()`. You may use any keys from the table to form the path. For example, consider a scenario where aggregate device states includes *inst1_frequency* of *915e6*, and *append()* has been called as *append(dut="DUT15")*. If the current aggregate state entry includes inst1_frequency=915e6, then the format string '{dut}/{inst1_frequency}' means relative data path 'DUT15/915e6'.

Parameters **format** – a string compatible with `str.format()`, with replacement fields defined from the keys from the current entry of results and aggregated states.

Returns None

set_relational_file_format(*format*)
Set the format to use for relational data files.

Parameters **format** (*str*) – one of 'csv', 'json', 'feather', or 'pickle'

set_row_preprocessor(*func*)
Define a function that is called to modify each pending data row before it is committed to disk. It should accept a single argument, a function or other callable that accepts a single argument (the row dictionary) and returns the dictionary modified for write to disk.

setup()
Open the file or database connection. This is an abstract base method (to be overridden by inheriting classes)

Returns None

write()
Commit any pending rows to the master database, converting non-scalar data to data files, and replacing their dictionary value with the relative path to the data file.

Returns the number of rows written

class labbench.data.**StatesToCSV**(*path, overwrite=False, text_relational_min=1024, force_relational=['host_log'], dirname_fmt='{id} {host_time}', nonscalar_file_type='csv', metadata_dirname='metadata', tar=False, **metadata*)
Bases: *labbench.data.StatesToRelationalTable*

Store data and states to disk into a master database formatted as a comma-separated value (CSV) file.

This extends *StateAggregator* to support

1. queuing aggregate state of devices by lists of dictionaries;
2. custom metadata in each queued aggregate state entry; and
3. custom response to non-scalar data (such as relational databasing).

Parameters

- **path** (*str*) – Base path to use for the master database
- **overwrite** (*bool*) – Whether to overwrite the master database if it exists (otherwise, append)
- **text_relational_min** – Text with at least this many characters is stored as a relational text file instead of directly in the database
- **force_relational** – A list of columns that should always be stored as relational data instead of directly in the database
- **nonscalar_file_type** – The data type to use in non-scalar (tabular, vector, etc.) relational data
- **metadata_dirname** – The name of the subdirectory that should be used to store metadata (device connection parameters, etc.)
- **tar** – Whether to store the relational data within directories in a tar file, instead of subdirectories

append(**args*, ***kwargs*)
Add a new row of data to the list of data that awaits write to disk.

This cache of pending data row is in the dictionary *self.pending*. Each row is represented as a dictionary of pairs formatted as {'column_name': 'row_value'}. These pairs come from a combination of 1) keyword arguments passed as *kwargs*, 2) a single dictionary argument, and/or 3) state traits configured automatically with *self.observe_states*.

The first pass at forming the row is the single dictionary argument

```
row = {'name1': value1, 'name2': value2, 'name3': value3}
db.append(row)
```

The second pass is to update with values as configured with *self.observe_states*.

Keyword arguments are passed in as

```
db.append(name1=value1, name2=value2, nameN=valueN)
```

Simple "scalar" database types like numbers, booleans, and strings are added directly to the table. Non-scalar or multidimensional values are stored in a separate file (as defined in *set_path_format()*), and the path to this file is stored in the table.

The row of data is appended to list of rows pending write to disk, *self.pending*. Nothing is written to disk until *write()*.

Parameters **copy=True** (*bool*) – When *True* (the default), use a deep copy of *data* to avoid problems with overwriting references to data if *data* is reused during test. This takes some extra time; set to *False* to skip this copy operation.

Returns the dictionary representation of the row added to *self.pending*.

clear()
Remove any queued data that has been added by append.

close()
Close the file or database connection. This is an abstract base method (to be overridden by inheriting classes)

Returns None

get()
Aggregate and return the current device states as configured with *observe()*.

Returns dictionary of aggregated states. Keys are strings defined by *key()* (defaults to '{device name}_{state name}'). Values are the type and value of the corresponding state of the device instance.

key(*device_name*, *state_name*)
Generate a name for a state based on the names of a device and one of its states or settings.

observe(*devices*, *changes=True*, *always=[]*, *never=[]*)
Deprecated - use *observe_states* instead

observe_settings(*devices, changes=True, always=[], never=['connected']*)
Configure Each time a device setting is set from python, intercept the value to include in the aggregate state.

Device may be a single device instance, or an several devices in an iterable (such as a list or tuple) to apply to each one.

Subsequent calls to *observe_settings()* replace the existing list of observed states for each device.

Parameters

- **devices** – Device instance or iterable of Device instances

- **changes** (*bool*) – Whether to automatically log each time a state is set for the supplied device(s)
- **always** – name (or iterable of multiple names) of settings to actively update on each call to get()
- **never** – name (or iterable of multiple names) of settings to exclude from aggregated result (overrides **:param:'always'**)

observe_states (*devices, changes=True, always=[], never=['connected']*)

Configure Each time a device state is set from python, intercept the value to include in the aggregate state.

Device may be a single device instance, or an several devices in an iterable (such as a list or tuple) to apply to each one.

Subsequent calls to *observe_states()* replace the existing list of observed states for each device.

Parameters

- **devices** – Device instance or iterable of Device instances
- **changes** (*bool*) – Whether to automatically log each time a state is set for the supplied device(s)
- **always** – name (or iterable of multiple names) of states to actively update on each call to get()
- **never** – name (or iterable of multiple names) of states to exclude from aggregated result (overrides **:param:'always'**)

open()

Instead of calling *open* directly, consider using *with* statements to guarantee proper disconnection if there is an error. For example, the following sets up a connected instance:

```
with StatesToCSV('my.csv') as db:
    ### do the data acquisition here
    pass
```

would instantiate a *StatesToCSV* instance, and also guarantee a final attempt to write unwritten data is written, and that the file is closed when exiting the *with* block, even if there is an exception.

set_device_labels(***mapping*)

Manually choose device name for a device instance.

Parameters **mapping** (*dict*) – name mapping, formatted as {device_object: 'device name'}

Returns None

set_path_format(*format*)

Set the path name convention for relational files that is used when a table entry contains non-scalar (multidimensional) information and will need to be stored in a separate file. The entry in the aggregate states table becomes the path to the file.

The format string follows the syntax of python's python's built-in `str.format()`. You may use any keys from the table to form the path. For example, consider a scenario where aggregate device states includes *inst1_frequency* of *915e6*, and *append()* has been called as *append(dut="DUT15")*. If the current aggregate state entry includes inst1_frequency=915e6, then the format string '{dut}/{inst1_frequency}' means relative data path 'DUT15/915e6'.

Parameters **format** – a string compatible with `str.format()`, with replacement fields defined from the keys from the current entry of results and aggregated states.

Returns None

set_relational_file_format(*format*)
Set the format to use for relational data files.

Parameters **format** (*str*) – one of 'csv', 'json', 'feather', or 'pickle'

set_row_preprocessor(*func*)
Define a function that is called to modify each pending data row before it is committed to disk. It should accept a single argument, a function or other callable that accepts a single argument (the row dictionary) and returns the dictionary modified for write to disk.

setup()
Open the file or database connection. This is an abstract base method (to be overridden by inheriting classes)

Returns None

write()
Commit any pending rows to the master database, converting non-scalar data to data files, and replacing their dictionary value with the relative path to the data file.

Returns the number of rows written

class labbench.data.**StatesToSQLite**(*path, overwrite=False, text_relational_min=1024, force_relational=['host_log'], dirname_fmt='{id} {host_time}', nonscalar_file_type='csv', metadata_dirname='metadata', tar=False, **metadata*)
Bases: *labbench.data.StatesToRelationalTable*

Store data and states to disk into an an sqlite master database.

This extends *StateAggregator* to support

1. queuing aggregate state of devices by lists of dictionaries;
2. custom metadata in each queued aggregate state entry; and
3. custom response to non-scalar data (such as relational databasing).

Parameters

- **path** (*str*) – Base path to use for the master database
- **overwrite** (*bool*) – Whether to overwrite the master database if it exists (otherwise, append)
- **text_relational_min** – Text with at least this many characters is stored as a relational text file instead of directly in the database
- **force_relational** – A list of columns that should always be stored as relational data instead of directly in the database
- **nonscalar_file_type** – The data type to use in non-scalar (tabular, vector, etc.) relational data
- **metadata_dirname** – The name of the subdirectory that should be used to store metadata (device connection parameters, etc.)
- **tar** – Whether to store the relational data within directories in a tar file, instead of subdirectories

append(**args*, ***kwargs*)
Add a new row of data to the list of data that awaits write to disk.

This cache of pending data row is in the dictionary *self.pending*. Each row is represented as a dictionary of pairs formatted as {'column_name': 'row_value'}. These pairs come from a combination of 1) keyword arguments passed as *kwargs*, 2) a single dictionary argument, and/or 3) state traits configured automatically with *self.observe_states*.

The first pass at forming the row is the single dictionary argument

```
row = {'name1': value1, 'name2': value2, 'name3': value3}
db.append(row)
```

The second pass is to update with values as configured with *self.observe_states*.

Keyword arguments are passed in as

```
db.append(name1=value1, name2=value2, nameN=valueN)
```

Simple "scalar" database types like numbers, booleans, and strings are added directly to the table. Non-scalar or multidimensional values are stored in a separate file (as defined in *set_path_format()*), and the path to this file is stored in the table.

The row of data is appended to list of rows pending write to disk, *self.pending*. Nothing is written to disk until *write()*.

Parameters **copy=True** (*bool*) – When *True* (the default), use a deep copy of *data* to avoid problems with overwriting references to data if *data* is reused during test. This takes some extra time; set to *False* to skip this copy operation.

Returns the dictionary representation of the row added to *self.pending*.

clear()
Remove any queued data that has been added by append.

close()
Close the file or database connection. This is an abstract base method (to be overridden by inheriting classes)

Returns None

get()
Aggregate and return the current device states as configured with *observe()*.

Returns dictionary of aggregated states. Keys are strings defined by *key()* (defaults to '{device name}_{state name}'). Values are the type and value of the corresponding state of the device instance.

key(*name*, *attr*)
The key determines the SQL column name. df.to_sql does not seem to support column names that include spaces

observe(*devices*, *changes=True*, *always=[]*, *never=[]*)
Deprecated - use *observe_states* instead

observe_settings(*devices, changes=True, always=[], never=['connected']*)
Configure Each time a device setting is set from python, intercept the value to include in the aggregate state.

Device may be a single device instance, or an several devices in an iterable (such as a list or tuple) to apply to each one.

Subsequent calls to *observe_settings()* replace the existing list of observed states for each device.

Parameters

- **devices** – Device instance or iterable of Device instances
- **changes** (*bool*) – Whether to automatically log each time a state is set for the supplied device(s)
- **always** – name (or iterable of multiple names) of settings to actively update on each call to get()
- **never** – name (or iterable of multiple names) of settings to exclude from aggregated result (overrides **:param:'always'**)

observe_states(*devices, changes=True, always=[], never=['connected']*)

Configure Each time a device state is set from python, intercept the value to include in the aggregate state.

Device may be a single device instance, or an several devices in an iterable (such as a list or tuple) to apply to each one.

Subsequent calls to *observe_states()* replace the existing list of observed states for each device.

Parameters

- **devices** – Device instance or iterable of Device instances
- **changes** (*bool*) – Whether to automatically log each time a state is set for the supplied device(s)
- **always** – name (or iterable of multiple names) of states to actively update on each call to get()
- **never** – name (or iterable of multiple names) of states to exclude from aggregated result (overrides **:param:'always'**)

open()

Instead of calling *open* directly, consider using *with* statements to guarantee proper disconnection if there is an error. For example, the following sets up a connected instance:

```
with StatesToSQLite('my.db') as db:
    ### do the data acquisition here
    pass
```

would instantiate a *StatesToCSV* instance, and also guarantee a final attempt to write unwritten data is written, and that the file is closed when exiting the *with* block, even if there is an exception.

set_device_labels(***mapping*)

Manually choose device name for a device instance.

Parameters **mapping** (*dict*) – name mapping, formatted as {device_object: 'device name'}

Returns None

set_path_format(*format*)

Set the path name convention for relational files that is used when a table entry contains non-scalar (multidimensional) information and will need to be stored in a separate file. The entry in the aggregate states table becomes the path to the file.

The format string follows the syntax of python's python's built-in `str.format()`. You may use any keys from the table to form the path. For example, consider a scenario where aggregate device states includes *inst1_frequency* of *915e6*, and *append()* has been called as *append(dut="DUT15")*. If the current aggregate state entry includes inst1_frequency=915e6, then the format string '{dut}/{inst1_frequency}' means relative data path 'DUT15/915e6'.

Parameters **format** – a string compatible with `str.format()`, with replacement fields defined from the keys from the current entry of results and aggregated states.

Returns None

set_relational_file_format(*format*)
Set the format to use for relational data files.

Parameters **format** (*str*) – one of 'csv', 'json', 'feather', or 'pickle'

set_row_preprocessor(*func*)
Define a function that is called to modify each pending data row before it is committed to disk. It should accept a single argument, a function or other callable that accepts a single argument (the row dictionary) and returns the dictionary modified for write to disk.

setup()
Open the file or database connection. This is an abstract base method (to be overridden by inheriting classes)

Returns None

write()
Commit any pending rows to the master database, converting non-scalar data to data files, and replacing their dictionary value with the relative path to the data file.

Returns the number of rows written

labbench.data.**read**(*path_or_buf*, *columns=None*, *nrows=None*, *format='auto'*, ***kws*)
Read tabular data from a file in one of various formats using pandas.

Parameters

- **path** (*str*) – path to the data file.
- **columns** – a column or iterable of multiple columns to return from the data file, or None (the default) to return all columns
- **nrows** – number of rows to read at the beginning of the table, or None (the default) to read all rows
- **format** (*str*) – data file format, one of ['pickle','feather','csv','json','csv'], or 'auto' (the default) to guess from the file extension
- **kws** – additional keyword arguments to pass to the pandas read_<ext> function matching the file extension

Returns pandas.DataFrame instance containing data read from file

labbench.data.**read_relational**(*path*, *expand_col*, *master_cols=None*, *target_cols=None*, *master_nrows=None*, *master_format='auto'*, *prepend_column_name=True*)
Flatten a relational database table by loading the table located each row of *master[expand_col]*. The value of each column in this row is copied to the loaded table. The columns in the resulting table generated on each row are downselected according to *master_cols* and *target_cols*. Each of the resulting tables is concatenated and returned.

The expanded dataframe may be very large, making downselecting a practical necessity in some scenarios.

TODO: Support for a list of expand_col?

Parameters

- **master** (*pandas.DataFrame*) – the master database, consisting of columns containing data and columns containing paths to data files
- **expand_col** (*str*) – the column in the master database containing paths to data files that should be expanded

- **master_cols** – a column (or array-like iterable of multiple columns) listing the master columns to include in the expanded dataframe, or None (the default) pass all columns from *master*
- **target_cols** – a column (or array-like iterable of multiple columns) listing the master columns to include in the expanded dataframe, or None (the default) to pass all columns loaded from each master[expand_col]
- **master_path** – a string containing the full path to the master database (to help find the relational files)
- **prepend_column_name** (*bool*) – whether to prepend the name of the expanded column from the master database

Returns the expanded dataframe

labbench.data.**to_feather**(*data*, *path*)

Write a dataframe to a feather file on disk. Any index will be moved to a column, index and column name metadata will be removed, and columns names will be changed to a string.

Parameters

- **data** – dataframe to write to disk
- **path** – path to file to write

Returns None

CHAPTER 4

labbench.host module

class labbench.host.**Host**(*resource=None*, ***settings*)

Bases: *labbench.core.Device*

> **Parameters** **resource** (Unicode *()*) – Addressing information needed to make a connection to a device. Type and format are determined by the subclass implementation

connect()

The host setup method tries to commit current changes to the tree

disconnect()

Backend implementations must overload this to disconnect an existing connection to the resource encapsulated in the object.

get_git_browse_url

Unicode(read_only=True)

get_git_commit

Unicode(read_only=True)

get_git_remote_url

Unicode(read_only=True,cache=True)

get_hostname

Unicode(read_only=True,cache=True)

get_log

Unicode(read_only=True)

get_time

Unicode(read_only=True)

metadata()

Generate the metadata associated with the host and python distribution

class settings(*device*, **args*, ***kws*)

Bases: labbench.core.HasSettingsTraits

Container for settings traits in a Device.

These settings are stored only on the host; setting or getting these values do not trigger live updates (or any communication) with the device. These define connection addressing information, communication settings, and options that only apply to implementing python support for the device.

The device uses this container to define the keyword options supported by its __init__ function. These are applied when you instantiate the device. After you instantiate the device, you can still change the setting with:

```
Device.settings.resource = 'insert-your-address-string-here'
```

trait attributes:

- *concurrency_support*: *Bool*
- *resource*: *Unicode*

concurrency_support
Bool(read_only=True)

Whether this backend supports threading

classmethod define(***kws*)
Change default values of the settings in parent settings, without redefining the full class. redefined according to each keyword argument. For example:

```
MyInstrumentClass.settings.define(parameter=7)
```

changes the default value of the *parameter* setting in *MyInstrumentClass.settings* to *7*. This is a convenience function to avoid completely redefining *parameter* if it was defined in a parent class of *MyInstrumentClass*.

resource
Unicode()

Addressing information needed to make a connection to a device. Type and format are determined by the subclass implementation

class state(*device*, **args*, ***kws*)
Bases: `labbench.core.state`

Container for state traits in a Device. Getting or setting state traits triggers live updates: communication with the device to get or set the value on the Device. Therefore, getting or setting state traits needs the device to be connected.

To set a state value inside the device, use normal python assigment:

```
device.state.parameter = value
```

To get a state value from the device, you can also use it as a normal python variable:

```
variable = device.state.parameter + 1
```

trait attributes:

- *connected*: *Bool*
- *git_browse_url*: *Unicode*
- *git_commit_id*: *Unicode*
- *git_remote_url*: *Unicode*

- *hostname*: *Unicode*
- *log*: *Unicode*
- *time*: *Unicode*

connected
Bool(read_only=True)

whether the `Device` instance is connected

classmethod getter (*func*)
Use this as a decorator to define a setter function for all traits in this class. The getter should take one argument: the instance of the trait to get. It should perform any operation needed to retrieve the current value of the device state corresponding to the supplied trait, using *self._device*.

One example is to send a command defined by trait.command.

The function should return a value that is the state from the device.

A trait that has its own getter defined will ignore this one.

git_browse_url
Unicode(read_only=True)

git_commit_id
Unicode(read_only=True)

git_remote_url
Unicode(read_only=True,cache=True)

hostname
Unicode(read_only=True,cache=True)

log
Unicode(read_only=True)

classmethod setter (*func*)
Use this as a decorator to define a setter function for all traits in this class. The setter should take two arguments: the instance of the trait to get, and the value to set. It should perform any operation needed to apply the given value to the trait's state in *self._device*. One example is to send a command defined by trait.command.

Any return value from the function is ignored.

A trait that has its own setter defined will ignore this one.

time
Unicode(read_only=True)

class labbench.host.**Email**(*resource=None*, ***settings*)
Bases: *labbench.core.Device*

Sends a notification message on disconnection. If an exception was thrown, this is a failure subject line with traceback information in the main body. Otherwise, the message is a success message in the subject line. Stderr is also sent.

Parameters

- **failure_message** (Unicode *()*) – subject line for test failure emails, or None to suppress success emails
- **recipients** (List *()*) – list of email addresses of recipients
- **resource** (TCPAddress *()*) – smtp server to use

- **sender** (Unicode *()*) – email address of the sender
- **success_message** (Unicode *()*) – subject line for test success emails, or None to suppress success emails

connect()
Backend implementations overload this to open a backend connection to the resource.

disconnect()
Backend implementations must overload this to disconnect an existing connection to the resource encapsulated in the object.

send_summary()
Sends the summary email containing the final state of the test.

class settings(*device*, **args*, ***kws*)
Bases: labbench.core.settings

Container for settings traits in a Device.

These settings are stored only on the host; setting or getting these values do not trigger live updates (or any communication) with the device. These define connection addressing information, communication settings, and options that only apply to implementing python support for the device.

The device uses this container to define the keyword options supported by its __init__ function. These are applied when you instantiate the device. After you instantiate the device, you can still change the setting with:

```
Device.settings.resource = 'insert-your-address-string-here'
```

trait attributes:

- *concurrency_support*: *Bool*
- *failure_message*: *Unicode*
- *recipients*: *List*
- *resource*: *TCPAddress*
- *sender*: *Unicode*
- *success_message*: *Unicode*

concurrency_support
Bool(read_only=True)

Whether this backend supports threading

classmethod define(***kws*)
Change default values of the settings in parent settings, without redefining the full class. redefined according to each keyword argument. For example:

```
MyInstrumentClass.settings.define(parameter=7)
```

changes the default value of the *parameter* setting in *MyInstrumentClass.settings* to *7*. This is a convenience function to avoid completely redefining *parameter* if it was defined in a parent class of *MyInstrumentClass*.

failure_message
Unicode()

subject line for test failure emails, or None to suppress success emails

recipients
List()

list of email addresses of recipients

resource
TCPAddress()

smtp server to use

sender
Unicode()

email address of the sender

success_message
Unicode()

subject line for test success emails, or None to suppress success emails

class state(*device*, **args*, ***kws*)
Bases: `labbench.core.HasStateTraits`

Container for state traits in a Device. Getting or setting state traits triggers live updates: communication with the device to get or set the value on the Device. Therefore, getting or setting state traits needs the device to be connected.

To set a state value inside the device, use normal python assigment:

```
device.state.parameter = value
```

To get a state value from the device, you can also use it as a normal python variable:

```
variable = device.state.parameter + 1
```

trait attributes:

- *connected*: *Bool*

connected
Bool(read_only=True)

whether the `Device` instance is connected

classmethod getter(*func*)
Use this as a decorator to define a setter function for all traits in this class. The getter should take one argument: the instance of the trait to get. It should perform any operation needed to retrieve the current value of the device state corresponding to the supplied trait, using *self._device*.

One example is to send a command defined by trait.command.

The function should return a value that is the state from the device.

A trait that has its own getter defined will ignore this one.

classmethod setter(*func*)
Use this as a decorator to define a setter function for all traits in this class. The setter should take two arguments: the instance of the trait to get, and the value to set. It should perform any operation needed to apply the given value to the trait's state in *self._device*. One example is to send a command defined by trait.command.

Any return value from the function is ignored.

A trait that has its own setter defined will ignore this one.

class labbench.host.**LogStderr**(*resource=None*, ***settings*)
Bases: *labbench.core.Device*

This "Device" logs a copy of messages on sys.stderr while connected.

Parameters **resource** (Unicode *()*) – Addressing information needed to make a connection to a device. Type and format are determined by the subclass implementation

connect()
Backend implementations overload this to open a backend connection to the resource.

disconnect()
Backend implementations must overload this to disconnect an existing connection to the resource encapsulated in the object.

class settings(*device*, **args*, ***kws*)
Bases: labbench.core.HasSettingsTraits

Container for settings traits in a Device.

These settings are stored only on the host; setting or getting these values do not trigger live updates (or any communication) with the device. These define connection addressing information, communication settings, and options that only apply to implementing python support for the device.

The device uses this container to define the keyword options supported by its __init__ function. These are applied when you instantiate the device. After you instantiate the device, you can still change the setting with:

```
Device.settings.resource = 'insert-your-address-string-here'
```

trait attributes:

- *concurrency_support*: *Bool*
- *resource*: *Unicode*

concurrency_support
Bool(read_only=True)

Whether this backend supports threading

classmethod define(***kws*)
Change default values of the settings in parent settings, without redefining the full class. redefined according to each keyword argument. For example:

```
MyInstrumentClass.settings.define(parameter=7)
```

changes the default value of the *parameter* setting in *MyInstrumentClass.settings* to *7*. This is a convenience function to avoid completely redefining *parameter* if it was defined in a parent class of *MyInstrumentClass*.

resource
Unicode()

Addressing information needed to make a connection to a device. Type and format are determined by the subclass implementation

class state(*device*, **args*, ***kws*)
Bases: labbench.core.HasStateTraits

Container for state traits in a Device. Getting or setting state traits triggers live updates: communication with the device to get or set the value on the Device. Therefore, getting or setting state traits needs the device to be connected.

To set a state value inside the device, use normal python assigment:

```
device.state.parameter = value
```

To get a state value from the device, you can also use it as a normal python variable:

```
variable = device.state.parameter + 1
```

trait attributes:

- *connected*: *Bool*

connected
Bool(read_only=True)

whether the `Device` instance is connected

classmethod getter (*func*)
Use this as a decorator to define a setter function for all traits in this class. The getter should take one argument: the instance of the trait to get. It should perform any operation needed to retrieve the current value of the device state corresponding to the supplied trait, using *self._device*.

One example is to send a command defined by trait.command.

The function should return a value that is the state from the device.

A trait that has its own getter defined will ignore this one.

classmethod setter (*func*)
Use this as a decorator to define a setter function for all traits in this class. The setter should take two arguments: the instance of the trait to get, and the value to set. It should perform any operation needed to apply the given value to the trait's state in *self._device*. One example is to send a command defined by trait.command.

Any return value from the function is ignored.

A trait that has its own setter defined will ignore this one.

CHAPTER 5

labbench.notebooks module

class labbench.notebooks.**panel**
Bases: object

Show tables summarizing device settings and states in jupyter notebook. Only a single panel will be shown in a python kernel.

Parameters

- **source** – Either an integer indicating how far up the calling tree to search for Device instances, or a *labbench.Testbed* instance.
- **ncols** – Maximum number of devices to show on each row

labbench.notebooks.**log_progress**(*sequence*, *every=None*, *size=None*, *title=None*)
Indicate slow progress through a long sequence.

This code is adapted here from https://github.com/alexanderkuk/log-progress where it was provided under the MIT license.

Parameters

- **sequence** – iterable to monitor
- **every** – the number of iterations to skip between updating the progress bar, or None to update all
- **size** – number of elements in the sequence (required only for generators with no length estimate)
- **title** – title text

Returns iterator that yields the elements of *sequence*

labbench.notebooks.**range**(**args*, ***kws*)
the same as python *range*, but with a progress bar representing progress iterating through the range

labbench.notebooks.**linspace**(**args*, ***kws*)
the same as numpy.linspace, but with a progress bar representing progress iterating through the range, and an optional title= keyword argument to set the title

CHAPTER 6

labbench.util module

labbench.util.**concurrently**(**objs*, ***kws*)

If **objs* are callable (like functions), call each of **objs* in concurrent threads. If **objs* are context managers (such as Device instances to be connected), enter each context in concurrent threads.

Multiple references to the same function in *objs* only result in one call. The *catch* and *flatten* arguments may be callables, in which case they are executed (and each flag value is treated as defaults).

Parameters

- **objs** – each argument may be a callable (function or class that defines a __call__ method), or context manager (such as a Device instance)
- **catch** – if *False* (the default), a *ConcurrentException* is raised if any of *funcs* raise an exception; otherwise, any remaining successful calls are returned as normal
- **flatten** – if not callable and evalues as True, updates the returned dictionary with the dictionary (instead of a nested dictionary)
- **nones** – if not callable and evalues as True, includes entries for calls that return None (default is False)
- **traceback_delay** – if *False*, immediately show traceback information on a thread exception; if *True* (the default), wait until all threads finish

Returns the values returned by each function

Return type dictionary of keyed by function

Here are some examples:

Example Call each function *myfunc1* and *myfunc2*, each with no arguments:

```
>>> def do_something_1 ():
>>>     time.sleep(0.5)
>>>     return 1
>>> def do_something_2 ():
>>>     time.sleep(1)
```

(continues on next page)

(continued from previous page)

```
>>>     return 2
>>> rets = concurrent(myfunc1, myfunc2)
>>> rets[do_something_1]
1
```

Example To pass arguments, use the Call wrapper

```
>>> def do_something_3 (a,b,c):
>>>     time.sleep(2)
>>>     return a,b,c
>>> rets = concurrent(myfunc1, Call(myfunc3,a,b,c=c))
>>> rets[do_something_3]
a, b, c
```

Caveats

- Because the calls are in different threads, not different processes, this should be used for IO-bound functions (not CPU-intensive functions).
- Be careful about thread safety.

When the callable object is a Device method, :func concurrency: checks the Device object state.concurrency_support for compatibility before execution. If this check returns *False*, this method raises a ConcurrentException.

labbench.util.**sequentially**(**funcs*, ***kws*)

Call each function or method listed in **funcs* sequentially. The goal is to emulate the behavior of the *concurrently* function, with some of the same support for updating result dictionaries.

Multiple references to the same function in **funcs* only result in one call. The *catch* and *flatten* arguments may be callables, in which case they are executed (and their values are treated as defaults).

Parameters **objs** – each argument may be a callable (function or class that

defines a __call__ method), or context manager (such as a Device instance) :param catch: if *False* (the default), a *ConcurrentException* israised if any of *funcs* raise an exception; otherwise, any remainingsuccessful calls are returned as normal :param flatten: if not callableand evalues as True, updates the returned dictionary with thedictionary (instead of a nested dictionary) :param nones: if notcallable and evalues as True, includes entries for calls that returnNone (default is False) :return: the values returned by each function :rtype: dictionary of keyed by function.

Here are some examples:

Example Call each function *myfunc1* and *myfunc2*, each with no arguments:

```
>>> import labbench as lb
>>> def do_something_1 ():
>>>     time.sleep(0.5)
>>>     return 1
>>> def do_something_2 ():
>>>     time.sleep(1)
>>>     return 2
>>> rets = lb.sequentially(myfunc1, myfunc2)
>>> rets[do_something_1]
1
```

Example To pass arguments, use the Call wrapper

```
>>> def do_something_3 (a,b,c):
>>>     time.sleep(2)
>>>     return a,b,c
>>> rets = lb.sequentially(myfunc1, Call(myfunc3,a,b,c=c))
>>> rets[do_something_3]
a, b, c
```

Because :func sequentially: does not use threading, it does not check whether a Device method supports concurrency before it runs.

class labbench.util.**Call**(*func*, **args*, ***kws*)
Bases: object

Wrap a function to apply arguments for threaded calls to *concurrently*. This can be passed in directly by a user in order to provide arguments; otherwise, it will automatically be wrapped inside *concurrently* to keep track of some call metadata during execution.

static cleanup(*func_in*)
Cleanup threading (concurrent execution only)

set_queue(*queue*)
Set the queue object used to communicate between threads

static setup(*func_in*)
Setup threading (concurrent execution only), including checks for whether a Device instance indicates it supports concurrent execution or not.

exception labbench.util.**ConcurrentException**
Bases: Exception

Raised on concurrency errors in *labbench.concurrently*

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class labbench.util.**ConfigStore**
Bases: object

Define dictionaries of configuration settings in subclasses of this object. Each dictionary should be an attribute of the subclass. The all() class method returns a flattened dictionary consisting of all values of these dictionary attributes, keyed according to '{attr_name}_{attr_key}', where {attr_name} is the name of the dictionary attribute and {attr_key} is the nested dictionary key.

classmethod all()
Return a dictionary of all attributes in the class

classmethod frame()
Return a pandas DataFrame containing all attributes in the class

class labbench.util.**ConcurrentRunner**
Bases: object

Concurrently runs all staticmethods or classmethods defined in the subclass.

This has been deprecated - don't use in new code.

class labbench.util.**FilenameDict**(**args*, ***kws*)
Bases: sortedcontainers.sorteddict.SortedDict

Sometimes instrument configuration file can be defined according to a combination of several test parameters.

This class provides a way of mapping these parameters to and from a filename string.

They keys are sorted alphabetically, just as in the underlying SortedDict.

clear()
Remove all items from sorted dict.

Runtime complexity: *O(n)*

copy()
Return a shallow copy of the sorted dict.

Runtime complexity: *O(n)*

Returns new sorted dict

classmethod from_filename(*filename*)
Convert from a FilenameDict filename string to a FilenameDict object.

classmethod from_index(*df*, *value=None*)
Make a FilenameDict where the keys are taken from df.index and the values are constant values provided.

classmethod fromkeys(*iterable*, *value=None*)
Return a new sorted dict initailized from *iterable* and *value*.

Items in the sorted dict have keys from *iterable* and values equal to *value*.

Runtime complexity: *O(n*log(n))*

Returns new sorted dict

get()
Return the value for key if key is in the dictionary, else default.

iloc
Cached reference of sorted keys view.

Deprecated in version 2 of Sorted Containers. Use `SortedDict.keys()` instead.

items()
Return new sorted items view of the sorted dict's items.

See `SortedItemsView` for details.

Returns new sorted items view

key
Function used to extract comparison key from keys.

Sorted dict compares keys directly when the key function is none.

keys()
Return new sorted keys view of the sorted dict's keys.

See `SortedKeysView` for details.

Returns new sorted keys view

peekitem(*index=-1*)
Return `(key, value)` pair at *index* in sorted dict.

Optional argument *index* defaults to -1, the last item in the sorted dict. Specify `index=0` for the first item in the sorted dict.

Unlike `SortedDict.popitem()`, the sorted dict is not modified.

If the *index* is out of range, raises `IndexError`.

Runtime complexity: *O(log(n))*

```
>>> sd = SortedDict({'a': 1, 'b': 2, 'c': 3})
>>> sd.peekitem()
('c', 3)
>>> sd.peekitem(0)
('a', 1)
>>> sd.peekitem(100)
Traceback (most recent call last):
  ...
IndexError: list index out of range
```

Parameters **index** (*int*) – index of item (default -1)

Returns key and value pair

Raises **IndexError** – if *index* out of range

pop(*key*, *default=<not-given>*)

Remove and return value for item identified by *key*.

If the *key* is not found then return *default* if given. If *default* is not given then raise `KeyError`.

Runtime complexity: *O(log(n))* – approximate.

```
>>> sd = SortedDict({'a': 1, 'b': 2, 'c': 3})
>>> sd.pop('c')
3
>>> sd.pop('z', 26)
26
>>> sd.pop('y')
Traceback (most recent call last):
  ...
KeyError: 'y'
```

Parameters

- **key** – *key* for item
- **default** – *default* value if key not found (optional)

Returns value for item

Raises **KeyError** – if *key* not found and *default* not given

popitem(*index=-1*)

Remove and return `(key, value)` pair at *index* from sorted dict.

Optional argument *index* defaults to -1, the last item in the sorted dict. Specify `index=0` for the first item in the sorted dict.

If the sorted dict is empty, raises `KeyError`.

If the *index* is out of range, raises `IndexError`.

Runtime complexity: *O(log(n))*

```
>>> sd = SortedDict({'a': 1, 'b': 2, 'c': 3})
>>> sd.popitem()
('c', 3)
>>> sd.popitem(0)
('a', 1)
```

(continues on next page)

(continued from previous page)

```
>>> sd.popitem(100)
Traceback (most recent call last):
  ...
IndexError: list index out of range
```

Parameters **index** (*int*) – *index* of item (default -1)

Returns key and value pair

Raises

- **KeyError** – if sorted dict is empty
- **IndexError** – if *index* out of range

setdefault(*key*, *default=None*)

Return value for item identified by *key* in sorted dict.

If *key* is in the sorted dict then return its value. If *key* is not in the sorted dict then insert *key* with value *default* and return *default*.

Optional argument *default* defaults to none.

Runtime complexity: *O(log(n))* – approximate.

```
>>> sd = SortedDict()
>>> sd.setdefault('a', 1)
1
>>> sd.setdefault('a', 10)
1
>>> sd
SortedDict({'a': 1})
```

Parameters

- **key** – key for item
- **default** – value for item (default None)

Returns value for item identified by *key*

update(**args*, ***kwargs*)

Update sorted dict with items from *args* and *kwargs*.

Overwrites existing items.

Optional arguments *args* and *kwargs* may be a mapping, an iterable of pairs or keyword arguments. See `SortedDict.__init__()` for details.

Parameters

- **args** – mapping or iterable of pairs
- **kwargs** – keyword arguments mapping

values()

Return new sorted values view of the sorted dict's values.

See `SortedValuesView` for details.

Returns new sorted values view

labbench.util.**hash_caller**(*call_depth=1*)

Use introspection to return an SHA224 hex digest of the caller, which is almost certainly unique to the combination of the caller source code and the arguments passed it.

labbench.util.**kill_by_name**(**names*)

Kill one or more running processes by the name(s) of matching binaries.

Parameters **names** (*str*) – list of names of processes to kill

Example

```
>>> # Kill any binaries called 'notepad.exe' or 'notepad2.exe'
>>> kill_by_name('notepad.exe', 'notepad2.exe')
```

Notes

Looks for a case-insensitive match against the Process.name() in the psutil library. Though psutil is cross-platform, the naming convention returned by name() is platform-dependent. In windows, for example, name() usually ends in '.exe'.

labbench.util.**check_master**()

Raise ThreadEndedByMaster if the master thread as requested this thread to end.

labbench.util.**retry**(*exception_or_exceptions*, *tries=4*, *delay=0*, *backoff=0*, *exception_func=<function <lambda>>*)

This decorator causes the function call to repeat, suppressing specified exception(s), until a maximum number of retries has been attempted. - If the function raises the exception the specified number of times, the underlying exception is raised. - Otherwise, return the result of the function call.

Example

The following retries the telnet connection 5 times on ConnectionRefusedError:

```
import telnetlib

# Retry a telnet connection 5 times if the telnet library raises␣
↪ConnectionRefusedError
@retry(ConnectionRefusedError, tries=5)
def connect(host, port):
    t = telnetlib.Telnet()
    t.open(host,port,5)
    return t
```

Inspired by https://github.com/saltycrane/retry-decorator which is released under the BSD license.

Parameters

- **exception_or_exceptions** – Exception (sub)class (or tuple of exception classes) to watch for
- **tries** (*int*) – number of times to try before giving up
- **delay** (*float*) – initial delay between retries in seconds
- **backoff** (*float*) – backoff to multiply to the delay for each retry
- **exception_func** (*callable*) – function to call on exception before the next retry

labbench.util.**show_messages**(*minimum_level*)

Configure screen debug message output for any messages as least as important as indicated by *level*.

Parameters **minimum_level** – One of 'debug', 'warning', 'error', or None. If None, there will be no output.

Returns None

labbench.util.**sleep**(*seconds*, *tick=1.0*)
Drop-in replacement for time.sleep that raises ConcurrentException if another thread requests that all threads stop.

labbench.util.**stopwatch**(*desc=''*)
Time a block of code using a with statement like this:

```
>>> with stopwatch('sleep statement'):
>>>     time.sleep(2)
sleep statement time elapsed 1.999s.
```

Parameters **desc** (*str*) – text for display that describes the event being timed

Returns context manager

class labbench.util.**Testbed**(*config=None*, *concurrent=True*)
Bases: object

Base class for Testbeds, which is a collection of multiple Device instances, database managers, etc. that together implement an automated experiment in the lab.

Use a *with* block with the testbed instance to connect everything at once like so:

```
with Testbed() as testbed:
    # use the testbed here
    pass
```

or optionally connect only a subset of devices like this:

```
testbed = Testbed()
with testbed.dev1, testbed.dev2:
    # use the testbed.dev1 and testbed.dev2 here
    pass
```

Make your own subclass of Testbed with a custom *make* method to define the Device or database manager instances, and a custom *startup* method to implement custom code to set up the testbed after each Device is connected.

after()
This is called automatically after disconnect, if no exceptions were raised.

cleanup()
This is called automatically immediately before disconnect if the testbed is connected using the *with* statement block.

Implement any custom code here in Testbed subclasses to implement startup of the testbed given connected Device instances.

make()
Implement this method in a subclass of Testbed. It should set drivers as attributes of the Testbed instance, for example:

```
self.dev1 = MyDevice()
```

This is called automatically when when the testbed class is instantiated.

startup()
This is called automatically after connect if the testbed is connected using the *with* statement block.

Implement any custom code here in Testbed subclasses to implement startup of the testbed given connected Device instances.

class labbench.util.**ThreadSandbox**(*factory*, *should_sandbox_func=None*)
Bases: object

Execute all calls in the class in a separate background thread. This is intended to work around challenges in threading wrapped win32com APIs.

Use it as follows:

obj = ThreadSandbox(MyClass(myclassarg, myclasskw=myclassvalue))

Then use *obj* as a normal MyClass instance.

exception labbench.util.**ThreadEndedByMaster**
Bases: RuntimeError

Raised in a thread to indicate the master thread requested termination

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

labbench.util.**until_timeout**(*exception_or_exceptions*, *timeout*, *delay=0*, *backoff=0*, *exception_func=<function <lambda>>*)
This decorator causes the function call to repeat, suppressing specified exception(s), until the specified timeout period has expired. - If the timeout expires, the underlying exception is raised. - Otherwise, return the result of the function call.

Inspired by https://github.com/saltycrane/retry-decorator which is released under the BSD license.

Example

The following retries the telnet connection for 5 seconds on ConnectionRefusedError:

```
import telnetlib

@until_timeout(ConnectionRefusedError, 5)
def connect(host, port):
    t = telnetlib.Telnet()
    t.open(host,port,5)
    return t
```

Parameters

- **exception_or_exceptions** – Exception (sub)class (or tuple of exception classes) to watch for
- **timeout** (*float*) – time in seconds to continue calling the decorated function while suppressing exception_or_exceptions
- **delay** (*float*) – initial delay between retries in seconds
- **backoff** (*float*) – backoff to multiply to the delay for each retry
- **exception_func** (*callable*) – function to call on exception before the next retry

Python Module Index

l